University of Alberta

SORTING USING SIMD REGISTERS

by

Timothy Michael Furtak ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2007

Your file Votre référence
ISBN: 978-0-494-33243-6
Our file Notre référence
ISBN: 978-0-494-33243-6

Abstract

Most contemporary processors offer some version of Single Instruction Multiple Data (SIMD) machinery — vector registers and instructions to manipulate data stored in such registers. The central idea of this thesis is to use these SIMD resources to improve the performance of the tail of recursive sorting algorithms by quickly sorting short sequences of elements. Data is loaded into the vector registers, manipulated in-register, and the result stored back to memory.

Three such sorting algorithms, as well as extensions for heapsort using d-heaps and for partitioning elements à la quicksort, are presented. Implementations on two different SIMD machineries — x86-64's SSE2 and G5's AltiVec — demonstrate that this idea delivers significant speed improvements. These improvements are orthogonal to the gains obtained through empirical search for a suitable sorting algorithm [21].

Table of Contents

List of Tables

List of Figures

List of Symbols

Sorting algorithms

Algorithm	Description
`DTSL`	DTSL's scalar sorting network, implemented using `if` statements.
`SSort`	SIMD sorting network which sorts 4 interleaved streams of equal length.
`ISort`	Insertion sort applied as a second pass after 1 or more SSort executions.
`MSort`	Mergesort applied to the streams from 1 or more SSort executions.
`RSort`	1-pass SIMD sorting network using realignment instructions.
`ISort`X	ISort algorithm with X streams.
`MSort`X	MSort algorithm with X streams.
`DTSL -` Y	DTSL's sorting network applied at Y threshold.
`ISort`X `-` Y	ISort algorithm with X streams applied at Y threshold.
`MSort`X `-` Y	MSort algorithm with X streams applied at Y threshold.
`RSort -` Y	One-pass RSort algorithm applied at Y threshold.
`Ins -` Y	Standard insertion sort applied at Y threshold.

Partitioning algorithms

Algorithm	Description
`S1`	Scalar binary partitioning function.
`S2`	Scalar ternary partitioning function.
`V1`	SIMD binary partitioning function.
`V2`	Algorithm V1 with an initial vectorized check for correctly partitioned elements at both ends of the array.

Chapter 1

Introduction

Sorting is a classical problem within computing science, with applications to a number of computational tasks. With the exception of such sorting algorithms as Bogosort[1] (and barring execution on a quantum computer), running time also tends to be a concern. Naturally, a large body of work exists relating to the topic of sorting algorithms and their execution on modern computers and theoretical computational models.

This thesis addresses the automatic generation of efficient code to sort short sequences of values. The idea is that an ahead-of-time optimizer searches for fast code for several sequence lengths and machine configurations. Then the compiler can simply instantiate such code when generating an optimized library. While algorithm-specific optimizations and empirical search have long been used both for scientific computation and for large parallel machines [8, 9, 34, 36], only recently these techniques were applied to integer-intensive, symbolic computation. Li *et al.* developed the Dynamically Tuned Sorting Library that adapts to the characteristics of the input to be sorted [21]. The main contribution of this thesis is the insight that the resources implemented in contemporary processors to enable SIMD computations can be put to good use to improve the performance of sorting short sequences. As demonstrated in this work, the effective use of these SIMD resources improves performance through the reduction of memory references, branches, and the increase in instruction level parallelism.

The initial inspiration for this work was the need for fast sorting of short sequences in the implementation of graphics rendering in interactive video-game applications. In such applications it is often necessary to decide, for each pixel of the image, what is the order of the elements that should be displayed [4]. Even though such Z-buffer (depth-buffer) pixel-ordering computations are typically handled by a specialized Graphics Processing Unit (GPU), partially transparent polygons must often first be sorted elsewhere, and there are plenty of similar ordering computations that are done

[1]Randomly permute the array of elements. Check if the array is sorted. Repeat while not.

A version of this chapter has been published.
Furtak, Amaral, Niewiadomski 2007. Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures. 348-357.

by the Central Processing Unit (CPU) in computer games. For instance, sorting is used to characterize the intensity of the various light sources that illuminate a character. Moreover, contemporary video-game application have at their disposal a rich supply of SIMD registers and instructions. For example, the PowerPC-based XBox 360 hardware features 128 AltiVec registers on each of its three cores along with an expanded set of AltiVec instructions. In addition to interactive video-game applications, sorting of short sequences is also present in particle-physics simulation applications.

Thus, using SIMD registers and instructions to sort small sequences is natural. Once a solution was created, applying it to the sequences that must be sorted at the tail-end of standard recursive sorting algorithms was the next logical step. The experimental evaluation of the vector-register-based sorting algorithms presented in this thesis use commodity processors (x86-64 and G5) and extensions to the DTSL library because these machines and algorithms are more readily available and exploitable than proprietary video-game hardware and software.

The algorithms presented are effective for sorting moderately-sized sequences of elements, as well as for sorting large sequences using quicksort. Additionally, we present SIMD algorithms for computing the index of a minimum (maximum) element. The elements being operated upon may be floating-point or integer values (keys), or "key-pointer pairs", comprised of a key and a memory address. The techniques for enabling the sorting of key-pointer pairs are then used to speed up *sift-down* operations in d-heaps, which are then used in the context of heapsort. These algorithms are described in Chapter 2. The integration of these sorting techniques with Quicksort, and SIMD functions for quicksort partitioning are discussed in Chapter 3. Finally, experimental results are given in Chapter 4 and conclusions in Chapter 5.

The main contributions of this thesis are:

- three algorithms that use the SIMD machinery of contemporary processors for efficient in-register sorting of short sequences;
- two partitioning algorithms that use SIMD instructions to split arrays into "low" and "high" sub-arrays;
- a method to use iterative-deepening search to find fast instruction sequences to move data within the SIMD registers;
- a method to use SIMD instructions to compute the minimum element in an array, with applications to d-heaps and heapsort;
- and an extensive experimental study on four different processors that demonstrate up to 30% improvement in the performance of quicksort for moderate- to large-sized arrays, and up to 20% in heapsort.

A preliminary version of this work appeared in [11]. Source code for the techniques within may be found online at [10].

1.1 Commodity Vector Hardware

Most contemporary processors offer some version of SIMD machinery — vector registers and instructions to manipulate data stored in such registers. Although the techniques and algorithms presented in this thesis may be applied to a fairly broad class of such vector machinery (modulo practical speed concerns), for concreteness we will restrict ourselves to the x86-64 SSE2 and the G5 AltiVec instruction set architectures.

These ISAs provide 128-bit wide vector registers. Depending on the instruction being executed, the data in these registers may be interpreted as 2 (64-bit) double-precision floating point numbers, 4 (32-bit) single-precision floating point numbers, 4 (32-bit) integers, etc. In keeping with SSE2, which offers fewer options for data types, we will be working with 32-bit granularity.

While specialized exceptions exist, the most common vector instructions perform the same logical/arithmetic operation element-wise between two vectors. For example, a vector add instruction would add the first element from one vector with the first element from another, similarly for the second elements, etc.

SSE2 provides some (slower) instructions for storing/loading to/from memory addresses not aligned on a 128-bit boundary, AltiVec does not. Misaligned loads using AltiVec must read two 128-bit vectors and then combine the desired elements via bit-/byte-wise rotation or some other data movement (shuffle) instructions.

1.1.1 G5 AltiVec

Compared to SSE2, the AltiVec data movement instructions have greater flexibility at the expense of increased setup requirements. Elements (bytes) from two source vectors may be copied anywhere within a third destination register (possibly one of the sources), but the instruction itself requires a fourth control vector to specify which source elements goes into which destination byte. Often this fourth vector is known at compile time and can simply be loaded from an array in memory, although for some cases it may/must be computed at runtime.

1.1.2 x86-64 SSE2

For our concerns, the majority of relevant x86-64 SSE2 instructions tend to be of a certain form. These instructions take one source vector and one source-cum-destination register. Unlike the AltiVec instructions which can select an arbitrary destination register distinct from the source(s), using SSE2 if both source vectors need to be preserved then a copy of one must be made.

As alluded to by AltiVec's flexibility, SSE2 shuffle instructions tend to be somewhat restrictive in terms of which elements can be moved where. There are however a number of such data movement instructions which cover common (de)interleaving operations, more so than for AltiVec. Thus a potential advantage is that control vectors for combining elements are unnecessary, either because the control is implicit in the instruction or it is given as an immediate operand in the assembly code.

One drawback with SSE2 is that instructions for combining elements from two vectors often place restrictions on how many elements from the source vectors must be selected (usually 2 from each), and where they can be placed in the destination register.

1.1.3 Floating Points and Integers

Although the AltiVec vector instructions provide support for both 32-bit floating point elements and 32-bit integers, SSE2 only support floating point values for certain arithmetic and comparison instructions that we wish to use, namely min/max and magnitude testing. However, due to how the IEEE Standard 754 floating point format is defined [15], for a large range of values (0 to $2^{31} - 2^{23} - 1$) unsigned integer encodings may be treated as floating point encodings (and vice versa) with respect to their ordering relations. This allows for somewhat limited (but potentially useful) support for using the following techniques to sort integers on x86-64 architectures using current SIMD instructions.

1.2 Related Work

Traditionally research for sorting on microprocessors has focused on two extremes. At one end there is classical sorting, operating on one machine with no allowance for parallelism. At the other end, one often assumes either a large number of distributed machines interconnected via some network topology, or a vector processor.

For instance, mesh-connected processor arrays have been well studied [32],[13]. In these models processors are arranged in a grid and communication is restricted to be between adjacent nodes.

The implementation of sorting in large-scale vector machines has also been extensively studied. Siegel produced one of the earliest descriptions of how to implement Batcher's sorting network, also known as *bitonic sorting*, in SIMD machines [31]. Bitton *et al.* provides an extensive description of such implementations [5]. The new contribution of this thesis is to demonstrate how the well-known sorting networks can be implemented in the SIMD machinery of contemporary processors and to indicate that code generators can instance such implementations to improve the performance of recursive sorting algorithms and heaps.

The idea of making better use of register resources within the processor to reduce the number of load or stores, in our case to put the SIMD resources to good use in sorting, is also explored by Arge *et al.* [35]. Their idea of forming cache-load-sized runs with quicksort is similar to our idea of switching to SIMD-register-based sorting at an appropriate threshold. The contrast is that we are also benefiting from the SIMD machinery which allows more parallelism in the execution and the elimination of branches, while they use the general-purpose registers and the storage available in a cache line.

Recently compilers have been used more often to improve the code generation for SIMD machinery in contemporary processors. Ren *et al.*'s approach of using an optimization algorithm to improve the data permutations is more general than our specific iterative-deepening search [29]. Nuzman *et al.* describes a compiler framework to generate vectorized code for interleaved data [23].

Additionally there has been a recent proliferation of commodity vector hardware in the form of GPUs and their programmable *shaders* for processing vertices and texture fragments. Facilitating this is increased GPU vendor support via software development kits for general purpose computations. Indeed GPUs have been used for matrix and FFT computations, dynamical simulations, and other numerically intensive tasks. In [27] Purcell *et al.* introduce using the GPU to implement Batcher's bitonic sort. Govindaraju *et al.* discuss using graphics processors for quantile and frequency estimation in [12], and to this end this also discuss using the GPU for sorting, but for much larger sets of data (databases 10s of gigabytes in size).

One of Govindaraju *et al.*'s methods is quite similar to our MSort in that data is streamed to the GPU, sorted, and then written back. Indeed, they also sort 4 streams of data in parallel, corresponding to one stream per color channel (plus alpha), and then merge the streams using the CPU. They note however that a practical concern in using the GPU is the current limitation in bus bandwidth for moving to and from the GPU — gains are achieved by leveraging the speed of computations and data movement within the GPU. They go on to implement more complicated sorting algorithms with good results. The existence and comparison of an SSE-optimized quicksort is mentioned in [12], although no details are provided as to where the SSE instructions are used or other implementation details.

The relationship between the SIMD-register-based sorting algorithms presented in this thesis and the development of DTSL is an orthogonal improvement to a library generator [21]. Li *et al.* focused on the dynamic identification of the best sorting algorithm for a given input sequence [22]. They selected an efficient algorithm for the tail of their recursive method. This thesis offers a better solution for the sorting of sequences that are small enough to benefit from the use of the SIMD machinery. Similarly, we provide a faster mechanism for selecting a minimum (maximum) child in the implicit d-heaps studied by LaMarca and Ladner [19, 20].

Our SIMD-register-based sorting could also improve partition based sorting methods. For instance, Shen and Ding use an adaptive partitioning scheme to attempt to evenly partition data into chunks smaller than cache size and then use quicksort or insertion sort to finish sorting each bucket [30]. This potential benefit is distinct from the actual partitioning phase itself, for which we also present a SIMD algorithm. Moreover, the dynamic analysis of the array's key distribution done by the DTSL may be used to select whether or not to use such vector partitioning.

This thesis offers an efficient solution for the sorting of sequences that are small enough to benefit from the use of the SIMD machinery, as well as algorithms for the large-scale partitioning of data and for the implementation of heaps.

Chapter 2

SIMD Vector Sorting Algorithms

Divide-and-conquer sorting algorithms such as quicksort often benefit from switching to a different sorting algorithm such as insertion sort when the number of elements is below a certain threshold. It is faster in these cases to call an algorithm with an asymptotically slower expect run time due to the constants involved. We will term such a sorting algorithm, called at the lowest levels of recursion, a "low-level" algorithm. By utilizing SIMD registers and instructions we will be able to construct low-level algorithms which are both fast and which can operate efficiently at much larger thresholds than insertion sort.

To implement such algorithms we first introduce sorting networks, then show how they can be made parallel using vector instructions. Two of these algorithms employ a second pass to finalize sorting, while the third uses data movement instructions, searched for ahead of time, to manipulate and sort all elements in-register. We then show how to extend these sorting algorithms to sort elements which are key-pointer pairs. Applying a similar technique, we create a vection function which can compute the index of a minimum (maximum) element, and use it within heapsort operations.

2.1 Sorting Networks

The inputs to an in-place *comparator*, $COMP(a,b)$, are two storage units — memory locations, registers, or vector-register elements — a and b, each containing a numerical input. After the comparator executes, the lower numerical value is stored in a and the higher numerical value is stored in b. Knuth describes a *comparator network* as a device that applies a fixed sequence of comparator operators to an input vector of a given size [18]. When a comparator network produces a sorted output for any possible input sequence, it is called a *sorting network*. The *size* of a sorting network is the total number of comparators in the network. The *depth* of a sorting network is the length of the critical path in its dependence graph. Therefore the depth provides a bound for the parallel execution of the sorting network, while the size provides a bound for a sequential execution.

A version of this chapter has been published.
Furtak, Amaral, Niewiadomski 2007. Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures. 348-357.

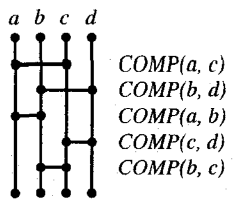


Figure 2.1: A 4-element sorting network.

An example of a sorting network with size 5 and depth 3 is shown in Fig. 2.1. The network is depicted as a set of value-carrying vertical rails and comparators. Values flow from top to bottom. A heavy dot at a line crossing indicates that the value at the vertical rail is an input to the comparator represented by the horizontal line. A comparator moves the larger value to the right, and the smaller value to the left. For instance, if the inputs are $a = 9, b = 5, c = 2, d = 7$, then the sorted output at the bottom of the sorting network is $a = 2$, $b = 5$, $c = 7$, and $d = 9$. The value 9 moves from rail a to rail c at $COMP(a,c)$, and then moves from rail c to rail d at $COMP(c,d)$.

Although several algorithms are available to generate code for sorting networks, Batcher's "odd-even mergesort" algorithm is often chosen for its efficiency [2]. Batcher's algorithm uses $O(n \log^2 n)$ comparators and has a depth of $O(\log^2 n)$. Sorting networks can be efficiently implemented in processors that provide a *min* and a *max* instruction. Sorting networks implemented with these instructions avoid the performance penalties of branch miss-predictions incurred by traditional branch-based sorting implementations. The experimental results in Section 4 indicate that eliminating branches in the code of sorting networks is a significant win in contemporary processors.

2.1.1 Supporting Hardware

Consider a machine that has the following *min* and *max* instructions:

$$min(a,b) = \begin{cases} a : a \leq b \\ b : otherwise \end{cases}, max(a,b) = \begin{cases} a : a \geq b \\ b : otherwise \end{cases}$$

The comparator required by a sorting network is easily constructed using these two operations, a copy instruction, and a temporary variable. For instance, such instructions are available in the x86-64 architectures supporting the SSE2 min and max operations that return the minimum (maximum) packed single-precision floating-point values [15].[1]

The extension of sorting networks to operate on vector instructions requires the definition of vectorized min and max instructions.[2] For input vectors A and B, $|A| = |B| = n$, let $C = min(A,B)$ be the element-wise minimum vector, such that $C_i = min(A_i, B_i)$ for $1 \leq i \leq n$. The vectorized max instruction is defined similarly. The *width* of a (vectorized) sorting network refers to the number of vectors being sorted. Given an ordered list of vectors $X^1, X^2, \ldots, X^n$, a *stream*

[1]SSE stands for Streaming SIMD Extensions. SSE2 improves upon the original SSE.
[2]These vector instructions are called a SIMD extension.

of data is defined to be the sequence formed by selecting the i^{th} element from each vector in order, such that the i^{th} stream is $X_i^1, X_i^2, \ldots, X_i^n$.

For instance, the x86-64 architecture has 16 XMM vector registers, and each register can hold 4 floating-point values. Therefore, sorting the values in n XMM registers using a sorting network produces 4 sorted streams of data of length n. Up to 15 XMM registers can be used, *i.e.* $1 \leq n < 16$, because one register must be reserved as temporary storage for the swap of values in the comparator.

This compare-and-swap machinery offers several advantages to sort a small set of values that fits within the SIMD registers: (1) its operation is unconditional and data independent; (2) it is inherently branch-free, and thus free of branch-prediction performance penalties; (3) it increases the bandwidth of sorting by enabling the SIMD instruction-level parallelism; and (4) each compare-and-swap requires the execution of only 3 instructions.

A code generator must be able to generate code to sort sequences of any length in a machine with $n+1$ SIMD registers. The solution is to define size-optimal sorting networks that use $1, 2, \ldots, n$ registers. The optimal code for the implementation of each of these sorting networks is pre-generated and stored in a small codebase available to the code generator for deployment. Once data has been loaded into the SIMD registers the code generator instantiates the code to perform the comparator operations specified by the sorting network, and integrates the resulting streams.

2.1.2 More Related Work

Sorting networks are a well-studied field and have a large body of associated research. Their asymptotic complexity is $O(n \log n)$ as established by Ajtai, Komlós, and Szemerédi in their famous work [1]. Unfortunately the constants involved in the AKS sorting network are prohibitively large for all practical purposes. As such, and as mentioned, it is Batcher's odd-even sorting network which is often chosen due to its small size.

Although it is not difficult to generate valid sorting networks, generating size-optimal sorting networks is quite difficult. At this time there are no known size-optimal sorting networks for more than 16 elements. Proving such optimality is a hard problem, as seen in [25] and the references therein.

In terms of hardware implementations, the number of interconnects required for a full sorting network may become overly large and/or expensive. Addressing this concern is the creation of periodic sorting networks, in which the output is cycled through a comparator network until it is sorted [16]. In a similar direction, [24] shows how to use a sorting network to sort arbitrarily large inputs.

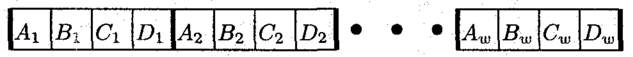
As mentioned in the previous chapter, sorting networks enjoy much popularity in GPU-based sorting methods. Batcher's other common and related sorting algorithm, Bitonic sort, also produces a sorting network. This sorting network and its variants are currently the algorithm of choice for implementing such GPU sorting.

2.2 Two-Pass Sorting with Streams

The first two SIMD-based sorting algorithms we will discuss operate in two phases. In the first phase the SIMD registers and instructions are used to generate a partially-sorted output. In the second phase a standard sorting algorithm — we investigate insertion sort and mergesort — finishes the sorting. The organization of data during the first phase will be dictated by the choice of algorithm for the second.

For the first phase, consider the use of the SIMD sorting machinery described in Chapter 2.1 for the task of sorting a sequence of $k \cdot w$ values using w SIMD registers, each register capable of storing k values. Each group of k values is loaded from memory into a separate SIMD register. For a moment, assume that the start of the sequence is aligned for such a load operation. The case of unaligned and/or non-multiple-of-k-sized arrays is discussed in Section 2.2.3.

The sorting machinery is then applied to produce k sorted streams of length w, and the sorted streams are written back in-place to memory in an interleaved form. After sorting, $A_1 \leq A_2 \leq \ldots \leq A_w$, $B_1 \leq B_2 \leq \ldots \leq B_w$, etc. For $k = 4$ the organization of the data in memory is as:



For convenience we will refer to this step/algorithm as SSort (where the "S" stands for *stream*). See Fig. 2.2 for SSort pseudocode. After this initial sorting pass the ordering relationship between elements from separate streams, A_i, B_j, C_k, and D_l, is still unknown. At this point the output from the vectorized sorting network must undergo an additional sorting pass.

2.2.1 Second Pass with Insertion Sort

A standard insertion-sort algorithm may be used to sort the output of the SIMD-based sorting network. Insertion sort performs best when its input is mostly sorted because the algorithm does not have to move elements very far. If the list is completely sorted, only a linear scan is needed. Thus a potential issue with using insertion sort as a second pass is how the data should be loaded into the SIMD vectors in the first phase to produce the most favorable input for insertion sort.

Consider an input sequence of S values, and a machine with $n + 1$ SIMD vectors (either physical or emulated). Each vector can store up to k values. Let $m = \lceil S/k \rceil$. If $m \leq n$ the entire array can be loaded into the SIMD registers, sorted, and written back in-place. Then a call to insertion sort will finish sorting the entire sequence.

If $m > n$, an in-place algorithm divides the array into subsets small enough to fit in the vector registers, sorts them with a sorting network, and writes each sorted subset back to the same locations.

A naïve approach would simply divide the array into $\lceil m/n \rceil$ almost equal-sized blocks. However, if the data is uniformly distributed this partition results in $\lceil m/n \rceil$ similar blocks, one after the other. The problem is that small elements from the last block would have similar values to the

```
Function SSort(X)
Data: Input array X with 4k elements.
begin
    $n \leftarrow |X|$
    $w \leftarrow n/4$
    for i=0 to $w-1$ do
        $V_i \leftarrow X[4i,\dots,4i+3]$                    /* Load into vector registers */
    end
    $S :=$ sorting network for $w$ elements (0-indexed)
    forall comparators $C\langle i,j\rangle$ in $S$ do
        $VCOMP(V_i,V_j)\begin{cases} COMP(V_i[0],V_j[0]) \\ COMP(V_i[1],V_j[1]) \\ COMP(V_i[2],V_j[2]) \\ COMP(V_i[3],V_j[3]) \end{cases}$
    end
    for i=0 to $w-1$ do
        $Y[4i,\dots,4i+3] \leftarrow V_i$                      /* Store into output array */
    end
    return $Y$                             /* Potentially the same array as X */
end
```

Figure 2.2: SIMD Stream sorting (SSort) algorithm.

small elements from the first block, and would require insertion sort to move many elements large distances across block boundaries.

A better approach is to load the blocks into the SIMD registers in a strided fashion. Consider for example $n = 4$ and $m = 12$ which requires three sorting network calls. Instead of the first call acting on vectors V^1, V^2, V^3, and V^4, it acts on V^1, V^4, V^7, and V^{10}. The second call acts on vectors V^2, V^5, V^8, and V^{11}, and the third on V^3, V^6, V^9, and V^{12}. In this way the small values in the array are likely to end up in V^1, V^2, and V^3. A stride width greater than one improves insertion sort performance in cases of uniform or mostly-sorted distributions.

The stride width may be passed as a parameter when calling SSort, although in our implementation it is hard-coded. We will call this strided version of the vectorized sorting network followed by an insertion sort pass ISort, shown in Fig. 2.3.

2.2.2 Second Pass with Mergesort

The mergesort algorithm, called MSort and shown in Fig. 2.4, uses a fixed-sized block of temporary storage T that is large enough to hold the entire array X. Because the SIMD-based sorting is applied to small sequences this array will not be large in practice. MSort proceeds as follows. Compute the number of blocks of data to be sorted, $\lceil m/n \rceil$, and allocate temporary space T. Call the sorting network on each block from X and store the sorted streams to T.

The *Q-MERGE* algorithm described by Wickremesinghe *et al.* [35] based on work by [28] is now used to store the sorted data into X: (1) Build a heap containing the first element in each stream, and

```
Function ISort(X, Y, w)
  Data: Input and output arrays of size $4k$, number of streams to use (multiple of 4).
  begin
    $n \leftarrow |X|$
    $p \leftarrow w/4$                                   /* Number of sorting passes */
    for $i$=1 to $p$ do
      $I_i := \{4(i-1)+j+w\cdot k\} : j \in \{0,1,2,3\};\quad k \in \mathbb{N};\quad 4(i-1)+j+w\cdot k < n$
      $X_i := X[I_i]$  /* Sub-array of staggered vector-sized blocks */
      $Y[I_i] \leftarrow$ SSort($X_i$)
    end
    InsertionSort($Y$, $n$)
  end
```

Figure 2.3: ISort sorting algorithm.

associate with each element a pointer to the next element in its stream; (2) Repeatedly extract the minimum element from the heap. During the extraction, replace the removed element with the next element in its stream, and rebuild the heap.

With a small number of streams, sufficient registers may be available to contain the entire heap. That is, the heap may be coded using explicit variables for each element, rather than a traditional array and index pointers. Maintenance operations for such small heaps must then be written using the fixed variable names/locations of elements, but in so doing they avoid potentially costly memory accesses and pointer indirections.

Heapify operations are then efficient and the only flow of data to/from memory is to fetch the next item from a stream or to store the next value to X. For heaps that are too large to fit within the available registers, in-memory heap code may be used. In practice one would most likely want to hard-code the heap using local variables and let the compiler automatically allocate them on the stack or within registers.

MSort uses one merge heap, with the number of inputs (*i.e.* heap elements) being a multiple of v, the number of elements in a vector register. That is, a heap completely handles the output from one or more vectorized sorting network (SSort) calls. The heap operations (build-heap, sift-down, extract-min), are hard-coded, with the heap source code produced automatically by a small program.

Additional optimizations include placing a sentinel value of infinity at the end of each stream to avoid checking if streams are empty [35]. Once a sentinel is loaded into the head it will sink to the bottom. When any sentinel is extracted from the heap no more non-sentinel values remain. If the elements being sorted may possibly contain values of infinity as their keys then it is straightforward after the main loop to extract the remaining elements by increment each stream pointer until it reaches the end of its stream. Since all remaining elements have the same key they will naturally be sorted.

Each sorting network call places elements from the same stream a constant distance away from each other. Thus the next element on a stream can be found by adding a constant offset to the

```
Function MSort(X, Y, w)
  Data: Input and output arrays of size $4k$, number of streams to use (multiple of 4).
  begin
    $n \leftarrow |X|$
    $p \leftarrow w/4$                                   /* Number of sorting passes */
    $b_0 \leftarrow 0$
    for $i$=1 to $p$ do
      $b_i \leftarrow \left\lfloor \frac{i \cdot n}{p} \right\rfloor$                 /* Boundary of sub-array $i$ */
      $n_i = b_i - b_{i-1} + 1$                          /* Size of pass $i$ */
      $Temp_i[0,\ldots,n_i - 1] \leftarrow$ SSort($X[b_{i-1},\ldots,b_i - 1]$)
      $Temp_i[n_i + 0] \leftarrow \infty$                /* Sentinel for first stream */
      $Temp_i[n_i + 1] \leftarrow \infty$                      /* Second stream, ... */
      $Temp_i[n_i + 2] \leftarrow \infty$
      $Temp_i[n_i + 3] \leftarrow \infty$
    end
    Build min-heap from the first element of each temporary stream.
    $i \leftarrow 0$
    while top of heap $\neq \infty$ do
      $Y[i] \leftarrow$ top of heap
      replace top element with the next element in its stream
      sift-down top element
      $i \leftarrow i + 1$
    end
    In case any "real" elements had a key of $\infty$:
    Finish copying elements to $Y$ from non-empty streams.
  end
```

Figure 2.4: MSort sorting algorithm.

address of the current element, which makes the maintenance of the "next element" pointer in the heap straightforward.

2.2.3 Unaligned Arrays

Potentially the arrays that we are sorting with these vector instructions are not aligned on vector boundaries (or do not have sizes that are exact multiples of 4). We will refer to the blocks of aligned elements in the middle of the array as the "core". On either end of the array, the elements that do not fill an entire block will be denoted as the "fringe". Likewise, the elements that are not part of the array proper, but are nevertheless located within the memory addresses of the blocks at either array end will be called the "non-array fringe" or "outer fringe".

In such cases where the outer fringe is non-empty, rather than only sorting the core elements and then attempting to shift the fringe elements into position[3] we will "pad" the outer fringe elements by placing values of positive and negative infinity into those locations as appropriate. Properly implemented, this will prevent elements in the outer fringe from moving when sorting is applied. After sorting, the original outer fringe values are restored.

[3]Or equivalently moving the proper elements into the fringe locations first.

```
Function VectorWrapper(Func, X, ...)
  Data: Sorting function (+parameters) and potentially unaligned input/output array.
  begin
    $n_1 \leftarrow$ LeftOutFringe($X$)   /* # of outer elements to X's left */
    $n_2 \leftarrow (-(|X| - n_1))$ positive mod4   /* # of right outer elements */
                                          /* Either 0, 1, 2, or 3 elements */
    for $i$=1 to $n_1$ do
      $L_i \leftarrow X[-i]$       /* Save elements in the left outer fringe */
      $X[-i] \leftarrow -\infty$
    end
    for $i$=1 to $n_2$ do
      $R_i \leftarrow X[n-1+i]$             /* Save the right outer fringe */
      $X[n-1+i] \leftarrow \infty$
    end
    Func($X, \ldots$)        /* Execution could have corrupted fringe */
    for $i$=1 to $n_1$ do
      $X[-i] \leftarrow L_i$            /* Restore original fringe values */
    end
    for $i$=1 to $n_2$ do
      $X[n-1+i] \leftarrow R_i$
    end
  end
```

Figure 2.5: Wrapper function to enable sorting of unaligned arrays using vector sorting functions.

The values stored in memory in the outer fringe may or may not be actual array elements, as opposed to arbitrary data. As such, care should be exercised that the values located in these positions will not change during the sorting, either due to parallel accesses or such locations actually holding local variables used during the sort. Alternately, as alluded, the fringe elements may be handled in a pre-/post-vector sort pass.

Note that the elements in the array proper must not move outside of that array into the fringes, as they will be lost and the array corrupted by padding elements. This is equivalent to requiring that the padding elements not move from their locations in the outer fringes, although reordering is allowed.

Negative sentinels from the start of the sequence will not shift their position during the SSort pass (using a stable comparator). By keeping track of the number of sentinels added, they may be removed from the start of their merge streams before building the heap. Similarly, during the final pass of MSort to extract real elements with value ∞, the true final index for each stream can be computed using the number of padding elements added.

Insertion sort's stability will automatically preserve the order of those first/last k elements which are equal (which naturally includes the sentinels).

We will refer to the use of a "wrapper function" for the process of correcting for misaligned arrays. For convenience we will exclude the specifics of passing fringe sizes to MSort and assume that such a wrapping function is ambivalent to the type of low-level algorithm it is containing.

2.3 One-Pass Sorting

The third SIMD-based sorting algorithm, which we shall call RSort, accomplishes the sorting in a single pass. Intuitively, this is possible by loading all of the n elements to be sorted into the vector registers, applying the comparators for an n-element (scalar) sorting network, and writing the elements back to memory in-place. In theory a vector-instruction-aware compiler would be able to automatically generate such a function from a scalar implementation of a sorting network.

As with the two-pass methods in Section 2.2, we can employ a wrapper function to handle the sorting of arrays that are not vector-aligned. Indeed the same wrapper function will work as listed if the RSort instructions make some minor assumptions discussed in section 2.3.3.

The difficulty with this RSort approach lies in relocating (aligning) elements within the vector registers such that the vector comparator operations can implement the scalar comparators. Depending on the fragmentation of free locations within the vector registers, this may be challenging. Moreover, since these comparators are indiscriminate, any "extra" elements in the input vectors may be corrupted and must be considered lost.

Since the cost of applying a vector comparator remains the same regardless of the number of "care" values in each input vector, a natural optimization is to execute more than one (scalar) sorting-network comparator at a time. However, the cost of additional data-movement instructions to properly position multiple comparator inputs in each vector register may outweigh the benefit of parallelization. In practice, for the sorting networks considered, it did not appear to be the case that aligning as many elements as possible[4] was ever detrimental to the resulting sequence of operations. That said, the algorithm we present does provide the ability to balance such alignment costs for the target architecture.

2.3.1 Searching to Align Vector Elements

We will first describe the algorithm used for finding a sequence of alignment instructions, and then show how this applies to a small 4-element sorting network.

Algorithm Input

The input to our algorithm is a sequence of comparators corresponding to a sorting network. In our case the sorting networks were produced by Batcher's *merge exchange* – not to be confused with Batcher's bitonic sort. Merge exchange has the convenient property of producing an initial sequence of comparators connecting elements that are separated by powers of 2. As the vectors are of length 4, the elements within are already aligned for this inital comparator sequence and require no repositioning.

[4]With the optimization that the sorting networks corresponding to Batcher's Merge Exchange are explicitly separated into layers.

The data dependencies in the sorting network define a partial ordering for the execution of the comparisons. The comparators can thus be partitioned into sets in such a way that all the comparators in each set can be executed in parallel. This partition corresponds to the computation of the maximal anti-chains in a data-dependency graph [33].

One natural optimization, namely considering multiple legal orderings of the comparator sequence, was not implemented due to the combinatorial increase in the search space. While we present no formal approximation bounds, we feel that the resulting suboptimality of the instruction sequences produced is not significant.

One important optimization which reduces both the number of assembly instructions and the time needed to search for an alignment sequence is to insert explicit breaks between levels of the merge exchange sorting network. That is, to disallow executing scalar comparators from different levels within one parallel comparator. For our purposes we consider levels to be the results of the innermost loop in Batcher's merge exchange algorithm as described in [18]. An 8-element merge exchange network is shown in Fig. 2.6 with such layer breaks indicated. One may also note that, as mentioned above, the elements are already aligned for executing the first 4 comparators in parallel.

Within a layer, a similar sequence of alignment instructions is often repeated for multiple parallel scalar comparators. Forcing breaks between levels may be thought of as helping to maintain this repeating pattern by not disrupting element alignment at the start of a layer. Such a disruption seems to propagate, requiring an increased number of alignment instructions.

Initial State

For convenience we will assume that we have an unbounded number of vector registers. The resulting sequence of assembly instructions can be restricted to a small number of physical vector registers as a post-processing step by "spilling" and loading values to and from memory as appropriate. This will be handled automatically by the compiler.

We will also assume that the elements are located in a continuous region of memory, begin on an appropriately aligned memory boundary, and that the number of elements is a multiple of the size of

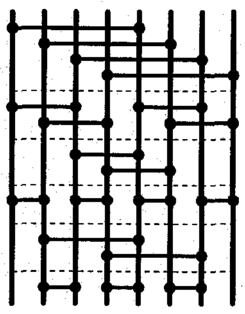


Figure 2.6: An 8-element sorting network produced by Batcher's merge exchange, with breaks indicated between layers.

a vector. These restrictions are for simplification only and can be removed by making small changes to the algorithm.

Note that the process of searching for a sequence of alignment instructions is only concerned with keeping track of the *labels* of the elements contained within the vector registers – we will refer to manipulations of elements only for convenience.

The first step is to load all of the elements from memory into the vector registers. It is natural and convenient to use a sequential labeling, such that the first memory location is labeled 0 and the last location $n - 1$.[5] Given realistic constraints on the capabilities of the vector manipulation instructions, a number of empty vector registers are required as swap space for rearranging elements. In our experiments having 5 empty vector registers in addition to those registers holding the initial values was seen to be sufficient. With appropriate pruning techniques during search, increasing this number should not seriously affect performance.

Aligning a Set of Comparators

While all of the comparators in the sorting network have not yet been executed, select the next k comparators that do not cross a layer and such that k is no larger than the number of elements in a vector register. The task is then to rearrange elements such that all the "low" elements from the k comparators are in one vector, and all the "high" elements in another,[6] and aligned element-wise with their partner.

Such an alignment is only valid if applying a vector comparator will not erase the last copy of any element. An erasure must necessarily occur when comparing an element with either an empty (garbage) value or another element with an unknown ordering relation.[7] Note that applying a vector comparator will also invalidate copies of compared elements that are located in other registers.

Finding a sequence of assembly instructions to accomplish this alignment is performed using a standard iterative-deepening search. The legal actions in a state are all vector assembly instructions which do not completely eliminate an element from the set of vector registers.

Due to feasibility concerns, each iterative-deepening search is divided into two phases: moving the low half of each comparator into one vector, and then moving the high half into alignment. If the maximum search depth in any one phase reaches 3, then that task is further subdivided into moving the first 2 elements into a vector, the next 2 elements into another, and finally combining them.

Even with these incremental stages, due to the massive branching factor a naïve implementation of this search would take a significant amount of time for even moderately large networks. Our implementation makes use of several heuristics to prune provably unproductive portions of the search

[5]The $0 \dots n - 1$ labeling is not required, but it does exploit the layout of Batcher's sorting networks.

[6]The partitioning of low and high elements can be dropped if a relabelling is performed on the fly during search, depending on whether or not the scalar comparator is inverted.

[7]We did not implement the extension where known orderings were exploited to preserve values. Namely, allowing an element to be compared against itself without consequence. Nor did we use other orderings which may be inferred from the partially executed sorting network.

space, given the current depth bound, such as detecting when the target values are spread out over too many vector registers to be combined within the number of remaining steps.

To address the tradeoff between the cost of executing a vector comparator and the cost of alignment instructions, the above search is repeated for smaller values of k, and the final cost becomes a combination of the number of alignment instructions and a penalty for including fewer scalar comparators than is possible. This penalty value is somewhat ad hoc, but should reflect the expected alignment and comparator costs for the current architecture.

Intuitively this attempts to select the sequence with the best ratio of number of alignments produced versus instructions required, with an additional bias towards producing more alignments since more alignments will reduce the total number of comparison steps.

When all appropriate values of k have been searched, the choice of how many comparators to include is made greedily and is not revisited. The vector comparator is then applied and the search continues using the remaining comparators.

Writing Values Back to Memory

After the final comparator has been applied the elements are sorted but are not (generally) located within the vector registers in an order in which they can be written back to memory. A similar iterative-deepening search now finds an instruction sequence to obtain the correct alignment. Naturally, all elements need not be aligned before any are written back to memory, although by using vector intrinsics the compiler should be able to order these writebacks any time after they become feasible.

2.3.2 Example Search

The sorting network shown in Fig. 2.1 will be used to illustrate the sequence of events in the alignment algorithm for single-pass in-register sorting. This network has four elements are requires the execution of five scalar comparators. An in-register sorting instance of this network using the x86-64 SSE(2) SIMD machinery is shown in Fig. 2.7. The instructions used in this instance are described in Table 2.1.[8]

The 4-element sorting network of Fig. 2.1 produces the following partitions:
$P_1 = \{COMP(a,c), COMP(b,d)\}$; $P_2 = \{COMP(a,b), COMP(c,d)\}$; and $P_3 = \{COMP(b,c)\}$.

First the elements of `XMM0` are assigned the four elements to be sorted (a, b, c, and d). Then a low-cost sequence of vector instructions is searched for to align a with c and b with d. Here this can be done with a single `movlhps` instruction in step 1. This allows for executing the $COMP(a,c)$ and $COMP(b,d)$ comparators in parallel (step 2).[9] After this comparison the value stored in element b

[8]Other SSE2 instructions frequently used for data movement but not included in this example are: `pshufd`, `unpckhps`, and `unpcklps`.

[9]For SSE2, a comparator between the contents of two registers `Ra` and `Rb` requires a temporary register `T` and the execution of three instructions: `movaps T, Ra`; `minps Ra, Rb`; and `maxps Rb, T`.

Instruction	**Description**
`movaps Ra, Rb`	copy the contents of `Ra` to `Rb`
`shufps Ra, Rb, i`	copy 2 elements of `Ra` to the 2 low-order words of `Ra`, and 2 elements of `Rb` to the 2 high-order words of `Ra`. The elements to be copied are specified by `i`.
`movhlps Ra, Rb`	copy the 2 high-order words from `Rb` to the 2 low-order words of `Ra`.
`movlhps Ra, Rb`	copy the 2 low-order words from `Rb` to the 2 high-order words of `Ra`.

Table 2.1: SSE2 instructions used in the example of Fig. 2.7

is smaller than the value stored in element d, and the value stored in element a is smaller than the value stored in element c.

In Fig. 2.7 a blank square represents a vector element that contains an unknown value that is not relevant to the sorting process. For instance, after the comparison in step 2 the values that were in elements b and a in the low-order words of `XMM0` may have moved. As they are not part of the sorting process they are now represented by blank squares. If the inputs to the sorting network are $a = 9$, $b = 5$, $c = 2$, and $d = 7$, this comparison would leave the highest-order words of `XMM0` and `XMM1` intact and would swap the contents of the second highest-order words. It may also swap the values in the two low-order words of these registers, but the contents of those words are irrelevant.

Now the two comparators in partition P_2 are candidates for the next vector alignment. The initial state for this search is the position of the elements in the vectors at the end of step 2. In the example in Fig. 2.7 a sequence of two instructions, `movhlps` and `shufps`, is selected to align elements d with c and b with a. Thus both comparators of P_2 can be executed in parallel in step 5.

A penultimate search is performed to execute the last comparator, resulting in steps 6 and 7, at which point the element values are sorted. Finally, the elements must be properly positioned within one register (in this case `XMM1`) before the sorted sequence can be written back to memory with a `movaps` instruction.

The vectorization of a sorting network only needs to be done once for each sorting network and for each architecture's set of vector instructions. Thus all the searches described above should be performed once and offline. The resulting schedule can then be used whenever a sequence of the corresponding size needs to be sorted.

2.3.3 Variants

At the start of this section we mentioned that we could use a wrapper function around RSort to correct for unaligned arrays. The function listed in Section 2.2 will work provided that the potential fringe elements (indices 0,1,2, n-3, n-2, and n-1) are not relabeled. This is to ensure that any sentinel values placed in these locations will not move when compared against an equal value.

On the matter of data alignment, and although not as fast as the basic RSort in practice, there are some potentially interesting modifications to the basic algorithm which may be of some use.

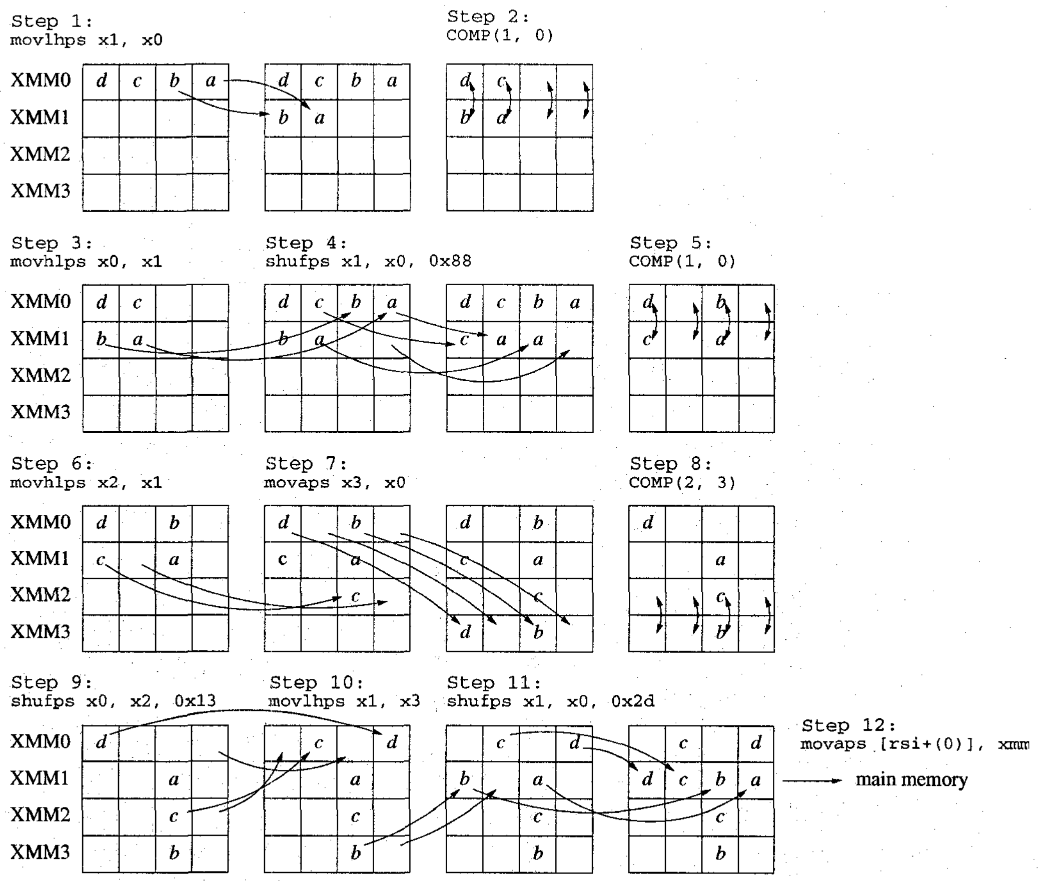


Figure 2.7: Instruction sequence to apply an in-register 4-element sorting network in an x86-64 architecture. The associated sorting network is shown in Fig. 2.1.

It is mainly for reasons of economy that we restrict ourselves to producing functions that assume aligned arrays. Relative to that baseline, it becomes considerably more expensive in terms of program size (with a subsequent execution penalty) to construct functions that deal with arrays which are not multiples of 4 nor are necessarily aligned.

Such instruction sequences can be constructed however, simply by properly initializing the register values in our alignment search. For the class of functions that operate on half-aligned arrays, say, leave the first two elements in the first vector blank One may wish to start counting on the first full vector to exploit the sorting network by setting the fringe values to the end — writing the elements back in sorted order rather than the initial labeling.

Also, when writing the sorted values back, it is important to either use operations that support copying only some vector elements, or to combine the sorted vector with the current fringe values before storing. Assuming that the fringe values will not be changed while the RSort call is executing, it is even possible to preserve them within the vector registers as part of the alignment steps.

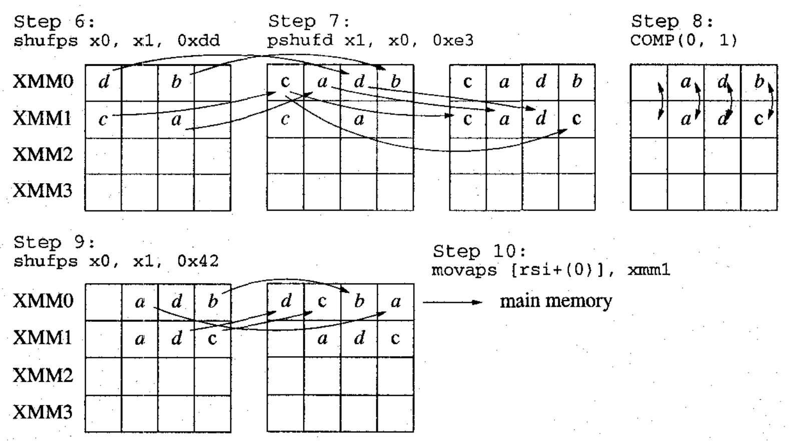


Figure 2.8: Alternate instruction sequence for the final steps of Fig. 2.7. Exploits comparing an element against itself. Yet another sequence could align the last comparator in one instruction, but this would require two instructions before writing back to memory.

2.4 Synchronous Data Movement

So far we have concerned ourselves only with the problem of sorting an array of floating-point values. A more general problem is that of sorting an array of data structures. To this end we will consider the problem of sorting 64-bit structures, where the first 32-bit word is a floating-point key, and the second 32-bit word is an associated pointer field. This second field may be an actual pointer to the element's full data structure in memory, an array index for the same, or possibly the full data itself.

For large data structures it is often more efficient to move only this key-pointer pair during sorting, rather than the full object. We will now describe the modifications necessary to allow the previously described vectorized sorting algorithms to operate on such key-pointer pairs.

The first requirement is to have the key values and the pointer values located in separate SIMD vectors, where the first element in the key vector is associated with the first element in the pointer vector, and so forth. This is accomplished by "swizzling" the keys and pointers when they are first loaded from memory and reversing this process when they are ultimately written back to memory.

Intermediate data movement and comparator operations may then be thought of as operating on one set of vector registers (the keys) in the same way as before, while those same data movements are duplicated on a second set of vector registers (the pointers). For unconditional data movements, such as alignment operations, this is straightforward to implement. Conditional data movement characterized by the operation of comparators, however, requires some care that the pointer elements in two vectors are exchanged if and only if a comparator exchanges their associated key values.

This conditional movement can be performed by predicating the exchange operations using a bitmask, rather than requiring branch instructions. Such predication is implemented via Boolean operations on the vector. The bitmask itself is generated by a vector (inequality) comparison operating on the elements in one key vector before and after applying a comparator. In this way we can produce a bitmask whose elements are all 1 where an exchange took place and 0 where one did not.

The AltiVec instruction set provides a vector instruction (`vsel`) specifically to perform such a combination of two vectors based on a bitmask. This is illustrated in Fig. 2.9. While a number of temporary copies are used in this example, an optimizing compiler can reduce the final number of instructions by relabeling variables. This is especially true since the function is only used inline, and such optimizations/relabelings may extend outside of the code listed. The x86-64 architectures have no comparable selection instruction — in this case we must resort to Boolean operations.

To this end an obvious, if slightly naïve, use of the bitmask (M) is to AND it with one vector (A), AND the other vector (B) with M's bitwise inverse, and then combine the two with an OR instruction. That is, $A' = (A \,\&\, M) \mid (B \,\&\, {\sim}M)$, and similarly for B'.

This is certainly correct, but, in terms of the number of instructions required, we can do better. Specifically, we can XOR A and B to produce a "swap" vector Q. By ANDing Q with M, we produce a vector that will exchange the required elements when $(Q \,\&\, M)$ is XOR'd with A and with B. This may be seen implemented using x86 GCC vector intrinsics in Fig. 2.10.

```
//================================================================

inline void compare_and_swap(vector float &a, vector float &b, // keys
                             vector float &A, vector float &B) // pointers
{
  vector float orig_a = a;
  vector float orig_b = b;
  vector float orig_A = A;
  vector float orig_B = B;

  a = vec_min(orig_a, orig_b);
  b = vec_max(orig_a, orig_b);

  vector int m = vec_cmpeq(a, orig_a);

  A = vec_sel(orig_B, orig_A, m);
  B = vec_sel(orig_A, orig_B, m);
}
```

Figure 2.9: Key-pointer comparator using GCC AltiVec vector intrinsics. Vectors `a` and `b` hold keys; vectors `A` and `B` hold the associated pointers. `m` is the selection mask.

Using this predicated data movement, we may return to the algorithm descriptions for SSort, ISort, MSort and RSort. These descriptions implicitly assumed that all of the data associated with an element would be properly handled by comparators and data movement instructions.

All of these algorithms can now correctly handle key-pointer pairs by replacing the standard key comparators with their key-pointer equivalents. Similarly for the unconditional data movement instructions, which, as mentioned, require only duplicating the data movement in a second set of (pointer) vector registers.

Such methods may be arbitrarily extended to structures larger than 64-bits, although the overhead of unpacking the elements and duplicating movement instructions (included predicated comparators) may quickly become prohibitive.

```
//==============================================================

inline void compare_and_swap(__m128 &a, __m128 &b, // keys
                             __m128 &A, __m128 &B) // pointers
{
  __m128 temp, M, Q;

  temp = a;
  a = _mm_min_ps(a,    b); // a' (the post-comparator value)
  b = _mm_max_ps(temp, b);

  M = _mm_cmpneq_ps(temp, a); // bitmask = 1 where elements were swapped (a != a')
  Q = _mm_xor_ps(A, B);
  Q = _mm_and_ps(Q, M);
  A = _mm_xor_ps(A, Q);
  B = _mm_xor_ps(B, Q);
}
```

Figure 2.10: Key-pointer comparator using x86 GCC vector intrinsics. Vectors a and b hold keys; vectors A and B hold the associated pointers. M and Q are the bitmask and XOR vector respectively.

2.5 Vectorizing ArgMin and ArgMax

The ArgMin (ArgMax) problem is the task of returning the parameter which corresponds to minimizing (maximizing) a given criteria. We will restrict ourselves to the sub-problem of finding the index of a minimum (maximum) array element. In this case the criteria being minimized (maximized) is the value of the element itself.

2.5.1 SIMD ArgMin/ArgMax

We present here a method for implementing ArgMin (ArgMax) by using SIMD vector instructions to compute the index of the child with minimum (maximum) key value.

This method is similar to the one used for sorting key-pointer pairs (mentioned in Section 2.4) in that it relies on the synchronous movement of values within a second set of registers. In this situation the values moving in synchrony are the *indices* of each element, such that for an array with n elements, the values range from 0 to $n-1$. The data read into the SIMD vectors from the element array is only used insofar as the key values condition the movement of the indices; the data is not written back and need not be preserved as before. For simplicity, assume that n is a multiple of k, the number of elements in a SIMD vector.

In the simplest implementation, such an ArgMin function is needed only for a small number of elements, and can be written as an unrolled loop. The indices being used are loaded from a constant and static array containing the values $0, 1, \ldots, n-1$.

The algorithm to determine the index of a minimum element proceeds as follows (for clarity, the loading/movement of the indices is implicit): (1) load the first k keys into one SIMD vector, call this register A; (2) while unread keys remain, read the next k keys into a SIMD vector B and set $A := min(A, B)$; repeat; (3) repeatedly halve the number of elements in A by taking the vector min of one half versus the other half, until only one element remains; (4) return the index of this element.

It is possible to construct a more general version of this function by computing the next set of indices to load on the fly. This can be done efficiently by adding a vector array of integer 1s to the previous iteration's indices. One may actually exploit the floating-point encoding to perform this operation on the x86-64 architectures by casting the unsigned integer data into float-point data.

```
Function ArgMin(A)
  Data: Input array of $4k$ elements.
  begin
    $n \leftarrow |A|$
    nblocks $\leftarrow n/4$
    $I := \{0,\ldots,n-1\}$
    $V_a \leftarrow A[0,\ldots,3]$
    $W_a \leftarrow I[0,\ldots,3]$
    for $i$=2 to nblocks do
      $V_b \leftarrow A[4(i-1),\ldots,4i-1]$
      $W_b \leftarrow I[4(i-1),\ldots,4i-1]$
      $M \leftarrow V_a < V_b$
      $W_a \leftarrow (W_a \wedge M) \vee (W_b \wedge \neg M)$
      $V_a \leftarrow$ Min($V_a$, $V_b$)
    end
    $M \leftarrow V_a[0,1] < V_a[2,3]$
    $W_a \leftarrow (W_a[0,1] \wedge M) \vee (W_a[2,3] \wedge \neg M)$
    $V_a \leftarrow$ Min($V_a[0,1]$, $V_a[2,3]$)
    $M \leftarrow V_a[0] < V_a[1]$
    $W_a \leftarrow (W_a[0] \wedge M) \vee (W_a[1] \wedge \neg M)$
    return $W_a[0]$
  end
```

2.5.2 Application to d-Heaps

d-heaps are a straightforward generalization of binary heaps where each internal node has $d \geq 2$ children. Increasing the value of d results in a shallower tree at the expense of requiring *sift-down* (a.k.a. *heapify-down*) operations to perform more comparisons when determining the child node with minimum (maximum) key value.

We will consider only implicit heap layouts, such that all elements are stored in a contiguous array, and parent-child relationships are determined by an element's index. The root node is located at index 0.[10] For a given node located at index i, the indices of its parent and child nodes can be easily computed using the following relations:

$$\texttt{parent}(i) = \lfloor \tfrac{i-1}{d} \rfloor$$

$$\texttt{child}(i,j) = i \cdot d + j; \quad 1 \leq j \leq d$$

In [19, 20] LaMarca and Ladner investigate the performance of implicit heaps and how they are affected by data caches. They suggest increasing the branching factor d as well as aligning the heap in memory such that the first child node begins on a cache-line boundary. The alignment with a cache-line is not critical for our purposes, but it tends to maximize the usage of each main memory

[10]Other common implementations use a starting index of 1.

access since all elements within a cache-line will be loaded into the cache when one of them is accessed.

We will employ our SIMD ArgMin function within the sift-down function to increase the branching factor that can be effectively used. For efficiency and simplicity we will restrict ourselves to values of d that are multiples of k. This has the significant benefit that if the root node is properly aligned then we are ensured that any node's children are located on SIMD vector boundaries. Proper alignment for the root node in this case is when it is located just before a SIMD vector boundary.

Calling ArgMin on the array of child nodes will return the offset from the first child. This value may then be added to the first child's index to get the index of a child with minimum key value. If the node being examined does not have d children (this can only occur at last internal heap node) then the vectorized search is replaced by a straightforward linear scan.

In situations where we have control over the allocation of the heap it is usually worthwhile to follow the suggestion in [19] and position the first child at the start of a cache-line – potentially requiring the allocation of a slightly larger array.

2.5.3 Heapsort

Heapsort operates by first turning an array of n elements into a max-heap. While the heap is non-empty, the current maximum (root) element is popped off, swapped with the last unsorted array location (*i.e.* the end of the implicit heap), and the heap property restored.

Although slower in practice on average than quicksort, heapsort has a running time of $\Theta(n \log n)$, while quicksort may degenerate to $O(n^2)$ behaviour. Heapsort also operates strictly in-place, whereas quicksort needs stack space to hold partition indices.

Exploiting our vectorized d-heap operations for heapsort is relatively straightforward, with the one concern that the child nodes must begin on a vector boundary. Rather than attempting to modify memory outside of the region to be sorted, we choose our root node to be the first such properly aligned location within the array to be sorted. This has the potential to exclude some number of elements at the start of the array from being sorted via the heap.

In the case of 64-bit key-pointer pairs and 128-bit SIMD vectors, at most 1 element may be so excluded by an unlucky alignment. This situation can be corrected in linear time by scanning through the array to locate the minimum element and then moving it to the first array location. This element is then guaranteed to be in its final sorted position. In general, if the number of elements which may be misaligned is a constant l, then a linear pass to find the l smallest elements requires $\Theta(n)$ time.

Variations

A somewhat common modification to the *sift-down* operation is given by Floyd. By default *sift-down* finds the smallest (largest) child node and compares its value to the current node. If this value is smaller (larger) then the child node moves up. When the value being sifted down has reached its proper location it is then written into the location just vacated by the child node that moved up.

Floyd's variation does away with comparing the child's key value against the parent's — the movement always takes place. When the value being sifted down reached the bottom of the heap, it is then sifted up. The benefit of this method is an expected reduction in the number of comparisons and exchanges.

However as noted in [19], once cache effects are taken into consideration, the frequency of cache misses can greatly overcome any savings thus gained. We implemented both methods.

For the first step in heapsort – building the heap – Floyd also proposes an alternative to constructing by adding one element at a time. The repeated adds method is intuitive in the sense that the next element being added is already located at the end of the heap and must only be sifted-up. Floyd's method here is to start at the middle layer of the heap and apply a sift-down operation for every node. This is repeated for the next layer up, terminating with the root. Again, fewer operations are performed at the price of data locality. Also again, we implement both methods.

2.6 Summary

We have presented three low-level sorting algorithms suitable for both stand-alone use and use within quicksort, in increasing order of implementation complexity. The first two employ a straightforward extension of scalar sorting networks, sorting 4 streams of data at a time. ISort concludes this pass with a call to insertion sort over the entire array, with the assumption that the first pass has resulted in a mostly- or partially-sorted array. MSort creates a merge heap to combine the sorted streams, and is far less affected by adversarial input. MSort requires more coding overhead than ISort, in terms of the explicit heap operations needed. RSort operates without any branches and has a running time that does not depend at all on the input. This comes at the expense of increased instruction code size and a greater implementation barrier as sequences of alignment instructions must be searched for.

Additionally, we have demonstrated how all three of these algorithms may be used to sort key-pointer pairs, and how this conditional data movement can be used to select the index of a minimum (maximum) element.

Chapter 3

Quicksort

Quicksort is the classic divide-and-conquer sorting algorithm described by Hoare [14]. As a divide-and-conquer algorithm, quicksort recursively partitions an input array into sub-arrays based on a pivot element, such that all elements which are less than the pivot are to its left, and all elements which are greater are to its right. Elements which have a key value equal to the pivot may be placed in either sub-array, although it is not uncommon for implementations to always place these elements into the left or the right sub-array.

This description refers only to the binary partitioning case. Ternary partitioning, a.k.a. multi-key quicksort, and equivalent to Dijkstra's *Dutch National Flag* problem [7], moves all elements equal to the pivot into one block inbetween the left and right sub-arrays.

We will use the non-recursive version of quicksort implemented by Li *et al.* in [21] which is quite similar to the version presented in Numerical Recipes in C [26]. This quicksort code maintains its own stack and always recurses into the smallest partition first – guaranteeing an $O(\log n)$ bound on the stack depth. Due to the explicit stack, if one wishes to detect pathological partitioning performance in terms of recursion depth – as a precursor to switching to a guaranteed $O(n \log n)$ algorithm – a secondary "depth stack" is needed to maintain a count of the recursion depth that would have resulted from a more traditional implementation.

3.1 Scalar Partitioning

In many performance analyses of sorting algorithms, execution time is estimated by the number of comparison operations. While this metric falls short of capturing the vagaries of the memory hierarchy and, to a lesser extent, architecture-specific instruction cycle counts and latencies, the number of comparisons still provides a useful time estimate.

Modern processors tend to pay a significant penalty for branch mispredictions. As comparisons are traditionally coupled with branch instructions, the number of comparisons tends to reflect both this misprediction penalty and the execution time of the surrounding code. Naturally, code may involve comparisons or branches which are not directly caused by comparisons between elements (such as loop conditions), although if the code is efficient, these will also be in an approximate 1-to-1 correspondence with the number of key comparisons.

As a fundamental property of comparison-based sorting methods, we will not be able to avoid a lower bound of $\Omega(n \log n)$ comparisons in general [6]. However we can strive to reduce the constant factors in these asymptotic bounds. Not to over-emphasize the importance of comparisons/branches at the expense of data movement, relocating elements is hardly a free operation on modern processors. Efficient partitioning code in particular is characterized by a tight loop containing mostly comparison and swap operations. To this end, the majority of quicksort/quickselect algorithm research has focused on reducing the number of comparison and copy instructions within partitioning algorithms while effectively choosing a good pivot. The result is a number of highly related algorithmic variants, often hinging on exploiting corner cases while supporting good performance on common input distributions.

A comprehensive analysis of partitioning algorithms is beyond the scope of this work — we will restrict ourselves to two common and efficient scalar variants for comparison. For those interested in saving every last compare/swap, Kiwiel [17] provides a detailed analysis of the instruction counts for several versions of binary and ternary partitioning schemes. For quicksort in general, one is directed to Bently and McIlroy [3].

3.1.1 Binary Partitions

As mentioned, there are some choices regarding how to go about partitioning – usually a matter of the order in which the array elements are examined. For binary partitioning one popular choice, shown in Fig. 3.1, (call this partitioning scheme *S1*) is to have two pointers starting from opposite ends of the array and moving towards each other. When the left pointer becomes greater than the pivot and the right pointer becomes less, the two elements are swapped. An alternate binary partitioning option given by Lomuto [3], is to walk from left to right, incrementally growing two sub-arrays by inserting (swapping) the next element into its proper position.

It should be noted that the two partitioning algorithms mentioned both assume that the pivot element is not located within the array this is being partitioned. That is, while elements with the same key value as the pivot may exist within the array, the element which was selected as the pivot during the quicksort recursive step has already been removed. Normally this is done by moving the pivot element to the start of the array, calling the partitioning algorithm, and then swapping the pivot element with the location corresponding to the end of the first sub-array.

As they stand, the two implementations have several tradeoffs depending on the distribution of the keys to be sorted. For a constant distribution (in which all keys and the pivot have the same value) S1 will split the array into two almost-equal sub-arrays, whereas Lomuto's method will degenerate to its worst-case quadratic time performance, moving all elements into one sub-array.

In the case of already sorted arrays S1 is significantly faster, and is slightly faster for a uniform distribution of keys.

```
//===================================================================

inline int partition_S1(DATA_T src[], int len, KEY_T pivot)
{
  int i = -1, j = len;

  while (1) {
    do i++; while (src[i].key < pivot);
    do j--; while (src[j].key > pivot);
    if (j < i) break;
    swap(src[i], src[j]);
  }

  return i; // index of last "small" element
}
```

Figure 3.1: Scalar binary partitioning function S1.

3.1.2 Ternary Partitions

As the number of distinct key values becomes small, ternary partitioning schemes tend to become faster. In general these methods require additional bookkeeping which is wasted if the number of elements equal to the pivot is small. Ideally the bookkeeping effort is recouped by not having to recurse on these equal elements. Unfortunately our experiments show that the extra effort is significant. As will be seen, unless there is a priori knowledge that the keys are drawn from a small distribution, or a statistical sample is gathered before sorting to support assumption, one is most likely better off using the binary partitioning function S1.

For our ternary partitioning implementation (call this function *S2*) we use the split-end partitioning function described in [3]. It operates much the same as S1 except an additional check is made for each element about to be swapped, to determine if it equals the pivot. If it does equal the pivot, it is moved to a sub-array of like elements at the start or end of the array. When all elements have been partitioned, these equal elements are moved from the edges of the array to the center.

3.2 Vector Optimizations

In a general sense, the vectorized partitioning function compares the keys of 4 elements at a time against the pivot. Based on the result of these comparisons, elements are then moved to the start or end of the array as appropriate.

More specifically, two writeback pointers are maintained, which are intialized to point to the start and end of the array. As elements are compared and written to their correct locations, these pointers advance towards each other. When the pointers cross, corresponding to (nearly) all elements having been written to their correct partition, the algorithm's main loop terminates. What we mean by "nearly" will be clarified shortly.

The writeback pointers correspond to "holes" in the array. The values previously located at these

```
//===================================================================

inline void partition_S2(DATA_T src[], int len, KEY_T pivot,
                         int &i, int &j)
{
  int a, b, c, d, s;

  // divide array into 5 regions:  [ (==) |a (<) b| (?) |c (>) d| (==) ]

  a = b = 0;
  c = d = len - 1;

  for (;;) {
    while (b <= c && src[b].key <= pivot) {
      if (src[b].key == pivot) { swap(src[a], src[b]); ++a; }
      ++b;
    }
    while (c >= b && src[c].key >= pivot) {
      if (src[c].key == pivot) { swap(src[d], src[c]); --d; }
      --c;
    }
    if (b > c) break;
    swap(src[b], src[c]);
    ++b;
    --c;
  }

  // move equal-to-pivot end regions into the middle

  s = min(a, b-a);
  for (int l = 0, h = b-s; s; --s) { swap(src[l], src[h]); ++l; ++h; }
  s = min(d-c, n-1-d);
  for (int l = b, h = n-s; s; --s) { swap(src[l], src[h]); ++l; ++h; }

  i = len - d + c;
  j = b - a;
}
```

Figure 3.2: Scalar ternary partitioning function S2.

positions have been loaded into temporary storage (within a vector), and are the elements which are being compared against the pivot. Consequently, each writeback operation is paired with a read operation from the same side of the array so as to maintain a hole for the next iteration.

Because all data movement is performed using vector instructions, the writeback operations are, in a sense, batch operations. If the number of elements to be written to one partition does not fill a vector register then more comparisons must be performed until sufficiently many elements are obtained (except possibly for the very last writeback).

Since we are concerned with key-pointer pairs, each vector read/write operation involves two elements.[1] Thus, we may have at most 1 element pending a writeback for each side of the array. Moreover, since we always compare 4 elements at a time, if one partition has a pending writeback, then so does the other. Such a situation occurs when there is a 1-3 split in the pivot comparisons, and will persist until another 1-3 split is encountered.

To avoid checking the status of a "pending" bit within every loop iteration, this information is encoded within the program state by means of two separate code blocks for the main loop – one

[1]For 64-bit structs and 4-element 128-bit vectors.

assuming that the bit is set, and the other assuming that the bit is not. Jump instructions move between the two blocks as appropriate.

The parallel computation of the pivot is accomplished by moving the keys from 4 elements into one vector and comparing that against a vector containing 4 copies of the pivot value. This results in a vector mask corresponding to the comparison done (say, all 1s where less than the pivot). Unlike the situation with sorting networks, here we can no longer perform predicated data movement – the destinations depend on the comparison results. Instead, the results of the 4 comparisons are combined into one 4-bit number which is used in a C "switch" statement. For the x86 ISA, the "move mask" (movmskps) instruction, which concatenates the high bits from each vector element and writes the result in a general purpose register, is used.

The AltiVec instruction set lacks a corresponding instruction, but may emulate the same functionality by AND-ing the comparison mask against a $\{8,4,2,1\}$ vector and performing a horizontal add of the result. Unfortunately in practice the overhead required for this emulation is greater than any benefits gained on the G5.

Each case in the switch statement corresponds to one of the 16 different layouts of high- and low-partition elements. The associated instruction code writes elements to their correct partition (possibly involving some realignment instructions to group high/low elements within the same vector), and reads the next elements to compare. These data movement instructions are quite straightforward. In the case of $1-3$ splits, as previously mentioned, control is transferred between the two loops via goto statements.

It should be noted that the switch statement is well-suited to being implemented as a jump table. That is, the 4-bit value may be used as a lookup in an array containing a pointer to the associated code for each switch-case. Optimizing compilers will use this implementation, barring architectural considerations.

The vectorized partitioning function is only applied to those elements which are well-aligned for vector access. The small number of "fringe" elements at the start and/or end of the array are left until the end when they are handled in a cleanup step. This is a simple matter of taking the large elements from the start fringe and moving them into the high partition – similarly for the small elements in the end fringe. This is depicted in Fig. 3.3. There will be 0 or 1 elements in the low fringe, and 0 to 3 elements in the high fringe, corresponding to half-vector-sized key-pointer elements and the requirement to compare four elements at a time. We shall denote this partitioning function V1.

As currently stated, the vectorized partitioning algorithm is vulnerable to the case where all (or a large number of) elements are equal to the pivot. With a fixed comparison function ($<$ or $\leq$) all elements will be moved into one partition. However by keeping track of how many elements have been written to each partition, and then selecting $<$ or $\leq$ so as to bias elements equal to the pivot towards the smaller partition, the resulting partitions are of almost equal size.

The smallest partition is guaranteed to be $4\lfloor (p+3)/8 \rfloor$, where p is the number of elements

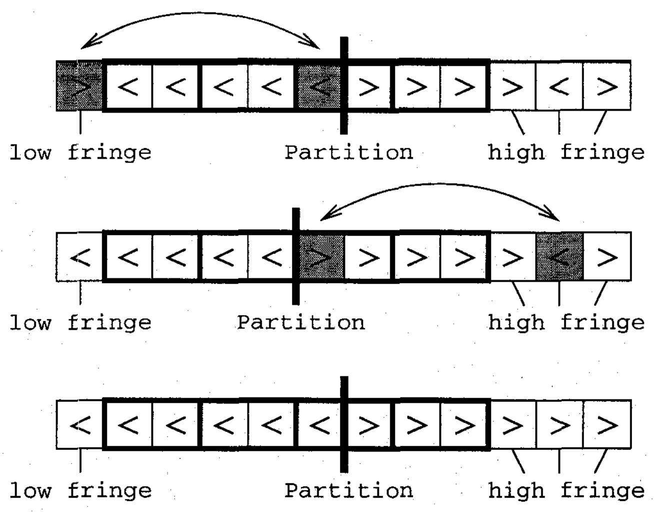


Figure 3.3: Sample cleanup step moving the fringe elements to their proper partition.

within the aligned array region that are equal to the pivot value. Clearly this should not be considered a remedy in general for a very bad or unlucky choice in selecting a pivot element.

3.2.1 Exploiting Special Cases

The scalar partitioning functions are able to take advantange of arrays that are already or mostly sorted by performing few swap operations – only when out-of-place elements are encountered. Unfortunately V1 will read and write every element, potentially even into its original location. In the interest of not having our vectorized function degrade on common inputs (or inputs that one might *expect* to sort quickly), we introduce a slight modification.

We will call this modified version V2, and the change is as follows. At the start of V1, the first 4 elements from the left side of the array are read in, and a vector comparison is performed against the pivot. If all elements are less than the pivot, then the necessary pointers are incremented and the next 4 elements are read in, otherwise stop scanning. Repeat this scan similarly for the right side of the array. Note that no elements need to be moved since we are only checking if they are already in their proper sub-arrays. Also, unlike the general case, here the G5 provides vector instructions to check if all comparisons are true or false.

This introduces some small overhead where either end of the array is not already in position, but has significant improvements for already sorted data, or reverse sorted data. Reverse sorted data will, in the first partitioning phase, divide into 2 (roughly) sorted sub-arrays if the median is selected as the pivot. "Roughly" in the sense that the exchange operations of V1 will, for efficiency, most likely not be implemented so as to exchange the order of element pairs when moving from the first half to the second and vice versa. The resulting key values might then be something along the lines of $1,0,3,2,5,4,\ldots$.

An adversarial-type input of a sorted array with the first and last elements exchanged will break this modification by halting the scans on the first step, while letting S1 operate with only one exchange. However we do not wish to get drawn into an "arms race" in designing special checks for our vector partitioning functions. We will simply reiterate that, as will be seen, V1/V2 operate well on common inputs.

3.3 Pivot Selection

To select a pivot element to use for partitioning, we employ Tukey's *ninther*, the median of 3 medians-of-3, as used in [3]. The result is intended to be a good approximation of the true median of the entire array being partitioned, while requiring few comparisons. The values used are taken from evenly-spaced locations within the array, with the first median-of-3 using the elements at indices 0, $\lfloor \frac{n-1}{8} \rfloor$, and $\lfloor \frac{2(n-1)}{8} \rfloor$. Similarly for the next two medians.

In practice this pivot selection is robust against common distributions, is fast, and does not require generating random numbers. Taking only the median of the first, middle, and last elements resulted in degenerate quicksort performance for the scalar binary partitioning function on reverse-sorted input arrays. Moreover, in general we expect the ninther to produce a better pivot value on average than only sampling three elements.

3.4 Thresholding

Now that we have established a sufficient library of vector sorting routines, we are ready to integrate them with quicksort. It is well established that quicksort should switch to a secondary (low-level) sorting algorithm like insertion sort when the number of elements drops below a certain threshold. Typically this is a small number, around 16 elements. We will simply replace the call to insertion sort with a call to one of our sorting functions (with some minor modifications to account for data alignment), although we must first determine what value(s) to use for thresholds.

The optimal threshold for a given low-level algorithm is going to depend on many related factors, although it is decided by how fast a low-level algorithm is able to sort arrays of increasing size relative to the partitioning speed. Clearly we want to find the point at which it becomes more expedient to sort an array with one of our algorithms rather than partition it further. This choice is going to depend on the specific architecture we are dealing with, the surrounding quicksort/partition code, and potentially the distribution of keys.

One general factor that will affect the ratio of partitioning time to low-level sorting is memory latency and bandwidth. Different algorithms will exhibit different memory access patterns, which may be a win if the given architecture is able to better support those requests — failure to do so results in memory stalls which can impose a serious performance penalty. Although being memory-bound may just indicate that the code has been well optimized. Assuming that the CPU is being provided with data at a sufficient rate, the efficiency of the code will be the main determinant of

```
Procedure WriteLow(a)
    $A[w_1, w_1+1] \leftarrow a$
    $a \leftarrow A[r_1, r_1+1]$
    $w_1 \leftarrow w_1 + 2$
    $r_1 \leftarrow r_1 + 2$
    $n_1 \leftarrow n_1 + 2$
```

```
Procedure WriteHigh(a)
    $A[w_2, w_2+1] \leftarrow a$
    $a \leftarrow A[r_2, r_2+1]$
    $w_2 \leftarrow w_2 - 2$
    $r_2 \leftarrow r_2 - 2$
    $n_2 \leftarrow n_2 + 2$
```

```
Function Partion_V1(A, p)
    Data: Input array of $4k \geq 8$ elements and the pivot value.
    begin
        $n \leftarrow |A|$
        $P := \{p,p,p,p\}$
        $r_1 \leftarrow 2$,  $r_2 \leftarrow n-4$                 /* low/high read index */
        $w_1 \leftarrow 0$,  $w_2 \leftarrow n-2$                /* low/high write index */
        $n_1$, $n_2 \leftarrow 0$                           /* low/high element count */
        $a \leftarrow A[0,1]$
        $z \leftarrow A[n-2, n-1]$
block0  while true do
            $V \leftarrow \{a,z\}$
            if $n_1 < n_2$ then $m \leftarrow$ 4-bit-mask($V \leq P$)
            else                $m \leftarrow$ 4-bit-mask($V < P$)
            switch m do
                WriteLow() and WriteHigh() as appropriate
                if $w_1 \geq w_2$ then goto finish
                if 1-3 split then
                    $t_1 \leftarrow$ low unpaired value
                    $t_2 \leftarrow$ high unpaired value
                    goto block1
                end
            end
        end
block1  while true do
            $V \leftarrow \{a,z\}$
            if $n_1 < n_2$ then $m \leftarrow$ 4-bit-mask($V \leq P$)
            else                $m \leftarrow$ 4-bit-mask($V < P$)
            switch m do
                WriteLow() and WriteHigh() as appropriate
                if $w_1 \geq w_2$ then goto finish
                if 1-3 split then
                    WriteLow({$t_1$, low unpaired value })
                    WriteHigh({$t_2$, high unpaired value })
                    goto block0
                end
            end
        end
finish  if no cached values in $t_1, t_2$ then return $w_1$
        write back cached values
        return $w_1 + 1$
    end
```

Figure 3.4: Vector binary partitioning function V1.

execution time.

While it would be possible to select a different threshold based on the size of the array or sampled distribution statistics, we will show that proper parameter selection depends mainly on the architecture and the partitioning algorithm. This simplifies implementation details somewhat, since we will be using a fixed partitioning scheme.

Rather than exhaustively testing on a range of common and adversarial key distributions, when selecting our threshold we restrict ourselves to the common cases in the hopes that they will generalize. Moreover, the effect of key distributions will mostly relate to the quality of partitioning. Distributions which produce consistently bad (*i.e.* uneven) partitions will approach worst-case quadratic time behaviour, at which point we may or may not cut our losses and switch to a guaranteed $\Theta(n \log n)$ algorithm (*e.g.* Heapsort). In either case, the low-level running time will not be a significant factor.

3.4.1 Searching for a Threshold

Our process for selecting an algorithm's threshold is as follows. For each architecture and each of partition functions S1, S2, and P1, we run quicksort on arrays of 5000, 10000, 20000, 40000, and 80000 elements, measuring the wall-clock time taken (or equivalently the number of processor cycles used, where such metrics are available).

We restrict our potential thresholds to multiples of 4, ranging from a lower value of 8 to an algorithm-specific large value. For ISort and MSort, the large value is the product of the number of streams each algorithm operates over and the maximum stream length capable of being sorted by SSort. We implemented the SSort parallel sorting networks for lengths of up to 64 elements (corresponding to 256 elements total). The maximum length for RSort is 256 elements. The DTSL scalar sorting network has a maximum length of 32 elements, and a cap of 256 elements was used for insertion sort. In all cases, the maximum value was selected so as to be inefficiently large. Namely, such that the most efficient threshold lies within the range considered.

The quicksort performance for each threshold is estimated using an average of 505 runs, discarding the first 5 runs. These runs are discarded to ensure that the relevant instructions and memory have had an opportunity to be loaded into processor's cache.

Due to time constraints, rather than evaluating all values in the range, a recursive search is performed. As an initialization step, using the threshold in the middle of the range, the quicksort performance time is sampled. We then iterate as follows. Starting from the threshold parameter which produces the best known evaluation time, two samples are taken using smaller and larger thresholds. The distance of these samples from the current best parameter begins at half the size of the threshold range (such that at the first step the two endpoints are sampled), and decreases by half with each iteration. This is analogous to performing a binary search. The quality of this search depends on the assumption that the performance curves for the thresholds are monotonically decreasing towards only one minima.

Finally, once the sample minimum has been found, to help mitigate any deviations of the sample points from the true values, the search finishes by searching all thresholds within a distance of 20 from the best parameter.

This parameter search is repeated multiple times until stability is achieved. That is, using the cached values from all the previous runs, a new search is started from the current best threshold, until no more new values are examined. In this way additional points will be sampled, as the best parameter from the last run will shift the center of the search away from the center of the array.

Once we have computed the best threshold for each of the 3 distributions – uniform, increasing, and decreasing – the overall best threshold is selected to be that which minimizes their weighted sum with coefficients of $\frac{1}{2}$, $\frac{1}{3}$, and $\frac{1}{4}$ respectively.

It will most likely be the case that we will have sample points for a given threshold in one distribution, but no matching sample for the same threshold in another distribution. In this case the execution time is taken to be the linear interpolation between samples. Since this interpolation may not accurately reflect the true value at that point, additional samples are taken to a distance of 20 around the current combined minimum for the uniform, increasing, and decreasing cases. As before, computing the combined threshold and taking samples around it is repeated until convergence.

The final threshold parameters we use for each algorithm/architecture/partitioning function are shown in Tables A.1, A.2, A.3, and A.4. One may note that in certain cases the selected threshold for the MSort and ISort functions can vary by a seemingly large amount across arrays of different sizes. This is due to those functions exhibiting exceptionally broad minima – a large range of threshold values will produce equally good performance results. This suggests that relatively little tuning is necessary for these algorithms. A representative sample of the threshold optimization curves are shown in Figs. 3.5, 3.6, and 3.7. While absolute execution times/cycle counts are included in these graphs, we hesitate to do so. At this point it is the shapes of the performance curves across array sizes and architectures that we wish to focus on. The timing results in Chapter 4 will be more comprehensive and will include the second vector partitioning function, V2.

We can see in Fig. 3.5 that on the Core 2 Duo MSort16, which merges 16 streams produced by 4 SSort passes, has a range of thresholds from at least 200 to 600 elements within which performance is essentially the same. The same pattern repeats for the other versions of MSort, based on the length of the streams, such that the curves for MSort32 would be spread over a larger range of thresholds. This indicates a fairly consistent scaling of the overhead from the merge heap relative to the total execution time. ISort exhibits similar scaling patterns and equally broad minima, shown in Fig. 3.7.

In the case of RSort, shown in Fig. 3.6, we can see a more convex minimum on the Core 2 Duo. In this case, however, the thresholds listed in Table A.1 are far more consistent. In all cases the choice of threshold tends to get quite near the optimal performance for all three curves. Thus, we may select a fixed threshold with a strong confidence that in most cases we could not have made a significantly better selection.

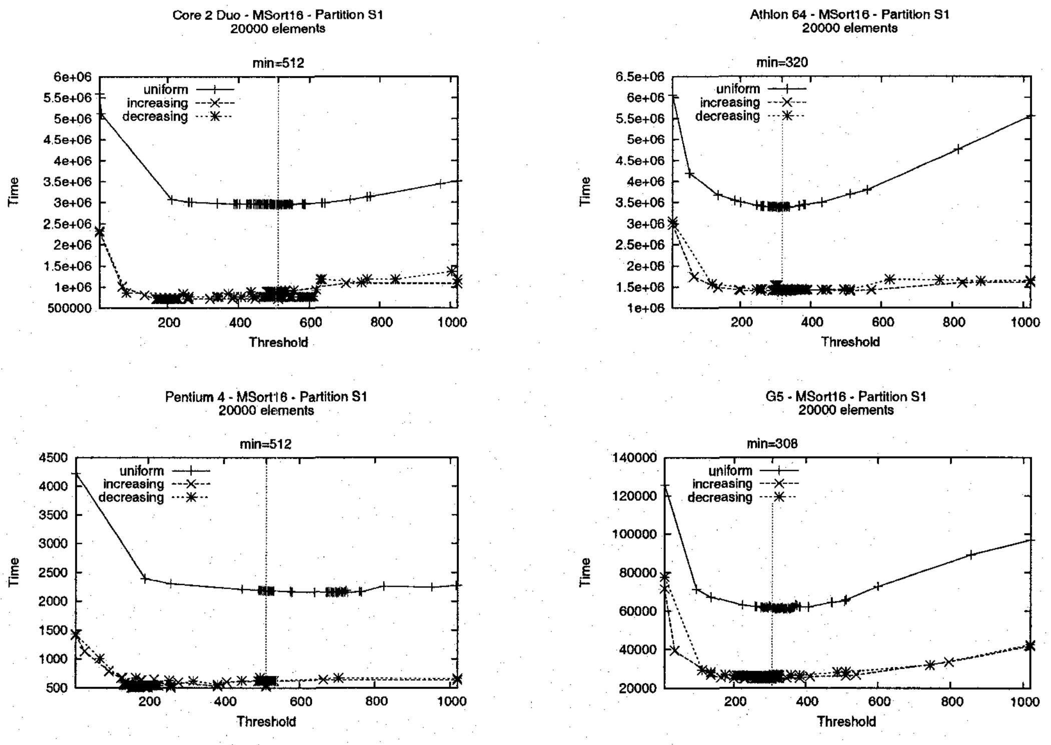


Figure 3.5: MSort16 thresholds for 20000 elements on different architectures.

3.5 Unaligned Arrays Redux

In Sections 2.2 and 3.1 we have previously discussed methods for wrapping our vector sorting and partitioning functions to support unaligned arrays. However if we are only concerned with using these algorithms within the context of quicksort then wrapping every execution is unnecessary. While we cannot ignore the fringes of the entire array, quicksort is still provably correct if we sort sub-arrays strictly larger than those returned by partitioning. As such, we can avoid some unnecessary instructions by not copying and restoring the fringes of the small sub-arrays.

Lemma 3.5.1. *Let $A[0, n-1]$ be an array of elements to sort. Let $A[x, y]$ be a sub-array produced by partitioning during quicksort, such that $\forall a$ with $x \leq a \leq y$ it is the case that $i < x \Rightarrow A[i] \leq A[a]$ and $y < i \Rightarrow A[a] \leq A[i]$. Then sorting $A[w, z]$ instead of $A[x, y]$ during quicksort, where $0 \leq w \leq x$ and $y \leq z < n$, results in A being correctly sorted when quicksort terminates.*

Proof. Without loss of generality we will only consider the elements in $A[w, x-1]$, since $A[y+1, z]$ may be treated analogously. Assume $w < x$. By virtue of having been partitioned, $A[i] \leq A[j]$ $\forall i, j : i < x \leq j$. Sorting $A[w, z]$ will not alter this relation, and if $A[w, x-1]$ has already been sorted, it will remain sorted. Moreover, if any elements in $A[w, x-1]$ are pivots (specifically a pivot separating $A[x, y]$ with the partition to its left), they will not change value. If $A[w, x-1]$

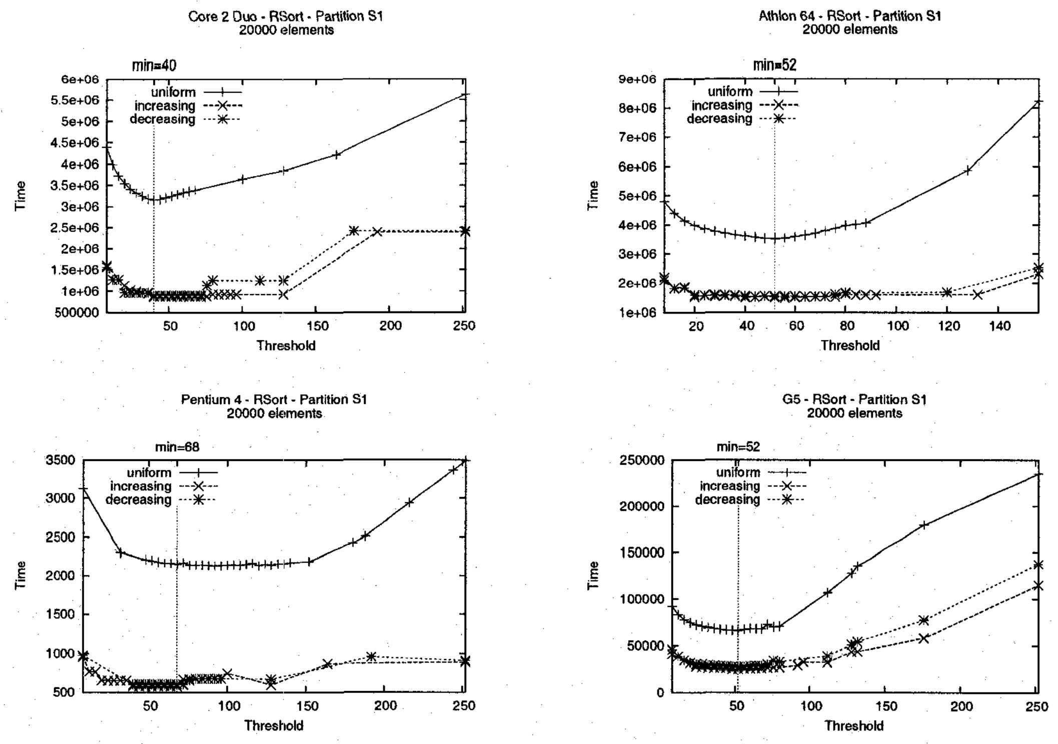


Figure 3.6: RSort thresholds for 20000 elements on different architectures.

has not yet been sorted, then becoming so will not affect the correctness of future sorting operations. This is clear since pivots will not move, and non-pivot values are not normally guaranteed to be in any order. Thus, regardless of the order in which the sub-arrays are considered during quicksort, A will be correctly sorted. $\square$

The extent to which this change improves performance will depend on the number of low-level sorting calls, which in turn depends on the size of the threshold. As such, RSort will benefit more than, say, MSort with many streams. While we did not perform extensive tests, on the Core 2 Duo there was a moderate improvement for RSort on the smallest arrays (200 elements) which became negligible as the array size increased to 500. MSort with 4 streams exhibited a slightly lesser gain. Neither improvement was sufficient to displace the fastest algorithm at that array size.

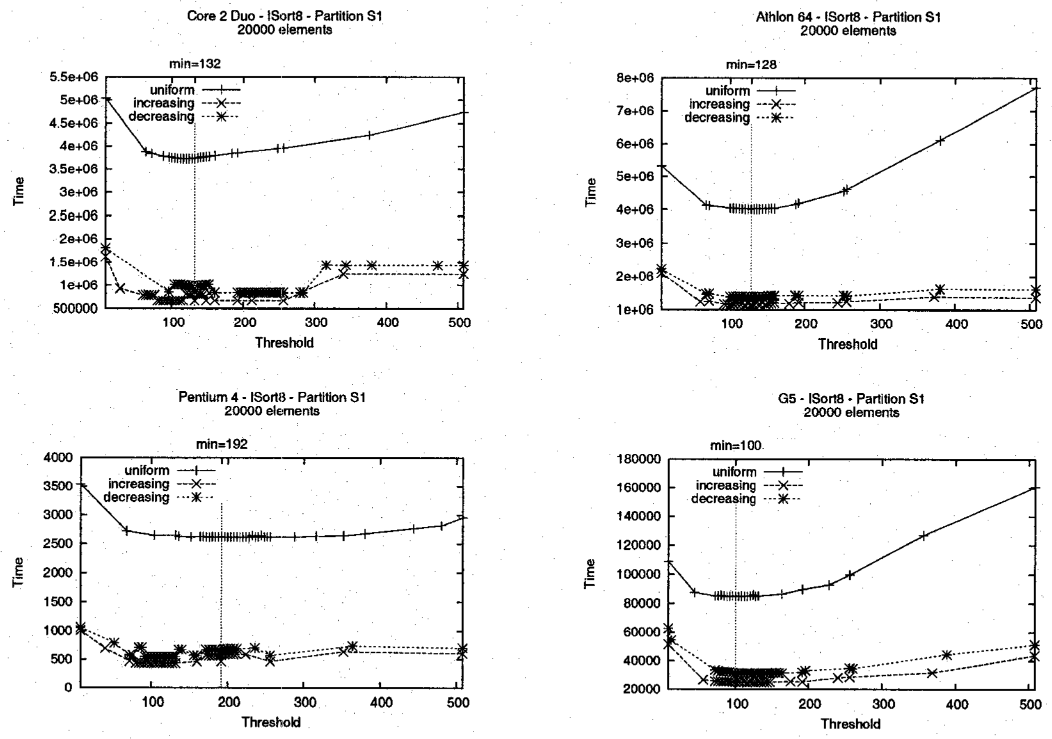


Figure 3.7: ISort8 thresholds for 20000 elements on different architectures.

Chapter 4

Experimental Evaluation

4.1 Overview

In this section we will be presenting detailed experimental evaluations of the low-level sorting algorithms discussed and Quicksort results using these low-level algorithms in combination with the scalar and vector partitioning functions described. Additionally, we present results using vectorized d-heaps in both a traditional heap setting and within the context of Heapsort for sorting an array of elements.

The main findings of these evaluations are:

- Significant reductions in execution time are possible for sorting on the Core 2 Duo, the Pentium 4, and the Athlon 64, with lesser reductions on the G5, depending on array size and distribution.
- The integration of SIMD-based sorting algorithms to sort sequences smaller than a fixed threshold can improve the performance of an optimized Quicksort when sorting 80,000-element arrays of floating-point key-pointer pairs by over 40%.
- Vectorized partitioning functions can further reduce execution time by about 15% over baseline, as arrays become larger and partitioning time becomes more dominant.
- These performance improvement are due not only to a reduction in the number of loads, stores, and branch instructions, but also to a significant decrease in the number of branch mispredictions.

4.1.1 Naming Conventions

For ease of reference, where appropriate within the figures, when we are referring to a low-level algorithm we will include the threshold at which that algorithm is applied. This is the same value which may be found by looking within the tables in Appendix A. For ISort and MSort, which combine a variable number of data streams, the number of streams will be included in the algorithm name. We may omit the number of streams if we are referring to the general performance for the class of MSort or ISort algorithms.

Algorithm	**Description**
`MSort`X - Y	MSort algorithm with X streams applied at Y threshold.
`ISort`X - Y	ISort algorithm with X streams applied at Y threshold.
`RSort` - Y	One-pass RSort algorithm applied at Y threshold.
`DTSL` - Y	DTSL's scalar sorting network applied at Y threshold.
`Ins` - Y	Standard insertion sort applied at Y threshold.

Table 4.1: Low-level vector sorting algorithms studied

4.1.2 Data

We will be referencing a significant number of graphs and tables within this chapter. In many cases a class of such figures (*e.g.* all G5 timings) will be too large to effectively insert within the main text. To avoid detracting from side-by-side comparisons of figures within a class we will direct the reader to the appropriate appendix, rather than splitting these figures up.

4.2 Hardware Specifics

Experiments were conducted on 4 different processor architectures: an E6400 Core 2 Duo, a 3.40GHz Pentium 4 (Prescott), an Athlon 64 3500+, and a PowerPC 970 G5. Where available, timing and event counts were taking using the processor's internal performance counters via the PAPI library. In other cases wall-clock times were used. An overview of each processor's cache sizes is given in Table 4.2. In all cases the benchmark applications did not require more memory than was physically available on the system.

Processor	**L1 I-Cache**	**L1 D-Cache**	**L2 Cache**	**SIMD Reg's**
Core 2	32K, 8-way	32K, 8-way	2M, 8-way, 64 byte lines	16
Pentium 4	12K μ-ops, 8-way	16K, 8-way	2M, 8-way, 64 byte lines	16
Athlon 64	64K, 2-way	64K, 2-way	512K, 16-way, 64 byte lines	16
G5	32K, 2-way	64K, 2-way	512K, 8-way, 128 byte lines	32

Table 4.2: Processor cache and register features.

In the case of the Core 2 Duo, the processor is overclocked to 3.2 GHz. There was no observed significant deviation in the results compared to running at stock speeds.

4.2.1 Software

All source code was written in C++ and was compiled using GCC 3.4.6, 4.0.1, 4.2.0, and 4.2.1 on the Pentium 4, G5, Core 2 Duo, and Athlon 64 respectively, with full optimizations and loop unrolling.

4.3 Low-Level Algorithm Timing

4.3.1 Methodology

To investigate the relative standings of our low-level timing algorithms for sorting short sequences of data, we ran each algorithm on its range of valid input sizes in steps of 4. Key values are drawn from a uniform distribution much larger than the array sizes, such that values are practically equivalent to a random reordering of $1 \ldots n$.

Performance for each size is taken as the average of 500 runs, with 5 "warm-up" runs to bring instruction code and data into the processor's cache. As such, we're measuring the best-case performance — in practice it may be the case that the instruction sequence for the function is not already cached, or the data being sorted needs to be fetched from main memory. Even when using the "same" algorithm (*e.g.* MSort, ISort) repeatedly, sorting a range of array sizes may use the instruction sequences from sufficiently many different sorting networks to push some of them out of the cache.

Although this is possible, the object file sizes for SSort and RSort listed in Table D.1 indicate that instruction code size may not be a serious problem if a program is repeatedly calling low-level SIMD sorting code for a small range of values (depending on the size of the cache). Naturally it is possible to construct artificial program execution and memory access patterns that can hide or amplify such an effect.

Note that for these timing results we are not running the algorithms inside a wrapper function (nor would insertion sort and the scalar sorting network require one). If the intended array size to sort is very small then such wrapping overhead may be a concern. We would however expect such cases to occur predominantly in situations where the array alignment could be explicitly controlled, either by aligning structures or manually locating data within padded regions.

4.3.2 Results

Core 2 Duo

Figure 4.1 shows the execution times for a representative sample of the low-level algorithms on very small arrays. In general the performance for ISort and MSort is slightly better when using a smaller number of streams for small arrays. RSort is significantly faster relative to all the other algorithms up until approximately 72 elements at which point MSort4 becomes faster. The scalar DTSL sorting network beats ISort12 and keeps pace with ISort4 up until 24 elements. Compared to the other algorithms, the scalar sorting network is only suited for a very small number of elements — the maximum size tested is only 32 elements. Larger sorting networks would not be particularly practical (given insertion sort as an alternative), and we expect them to scale poorly, considering their increasing code size and branch-intensive comparators.

As expected, insertion sort does performs poorly, scaling quadratically. These times for insertion sort would change significantly if the key values were already sorted, or sorted in reverse order, exhibiting better and worse performance respectively.

For moderately-sized arrays (~100 to 1000 elements) MSort is a clear winner. Figure 4.2 shows some of the MSort versions with different stream counts. Insertion sort and ISort12 are included to indicate relative performance. A notable feature of all the MSort versions is a single sharp jump in execution time corresponding to the point at which the SSort pass switches from sorting 40-element streams to 44-element streams.

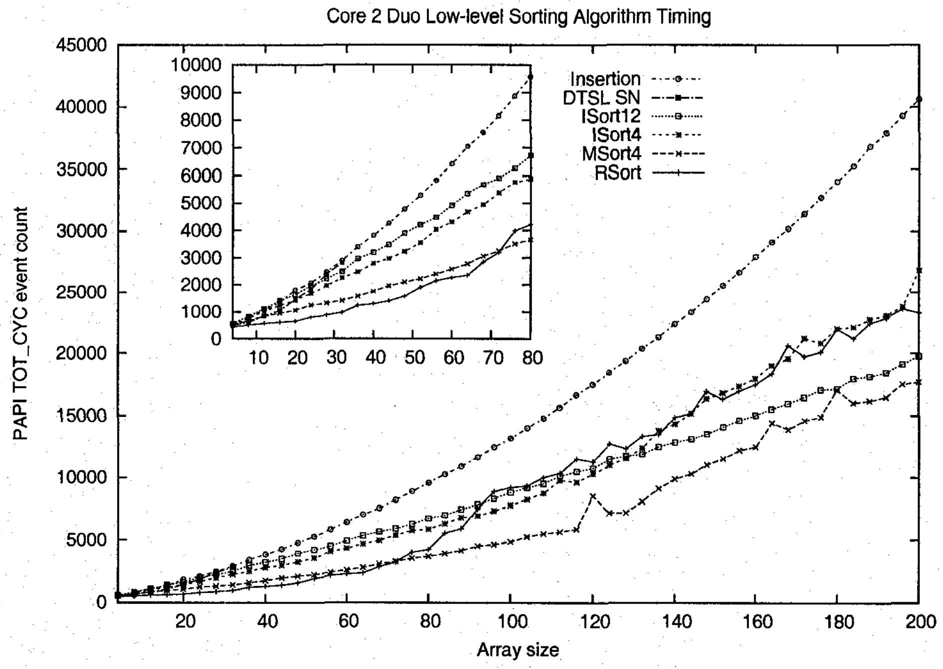


Figure 4.1: Low-level algorithm cycle counts on the Core 2 Duo.

Athlon 64 and G5

Performance on the Athlon 64 takes somewhat of a departure in that the cycle counts are much noisier than on the Core 2 Duo, as shown in in Fig. 4.3. ISort12 and MSort32 are effectively tied as the winner for small arrays. Results stabilize somewhat for moderately-sized arrays, with MSort32 taking the lead. Here, however, looking at Fig. 4.4, instead of the Core 2 Duo's discrete jump in running times we can observe two well-defined alternating execution profiles which are gradually diverging.

Specifically, for a given instance of MSort$X$ there is a dip in execution time every X elements, corresponding to the point at which all streams being sorting in the first pass are the same length. This would indicate that the instruction sequences for 2 sorting networks (being used when the streams are of non-uniform length) are pushing each other out of the cache. This same behaviour can be observed for the G5 in Fig. 4.5, which also has an L2 cache of only 512K, compared to the

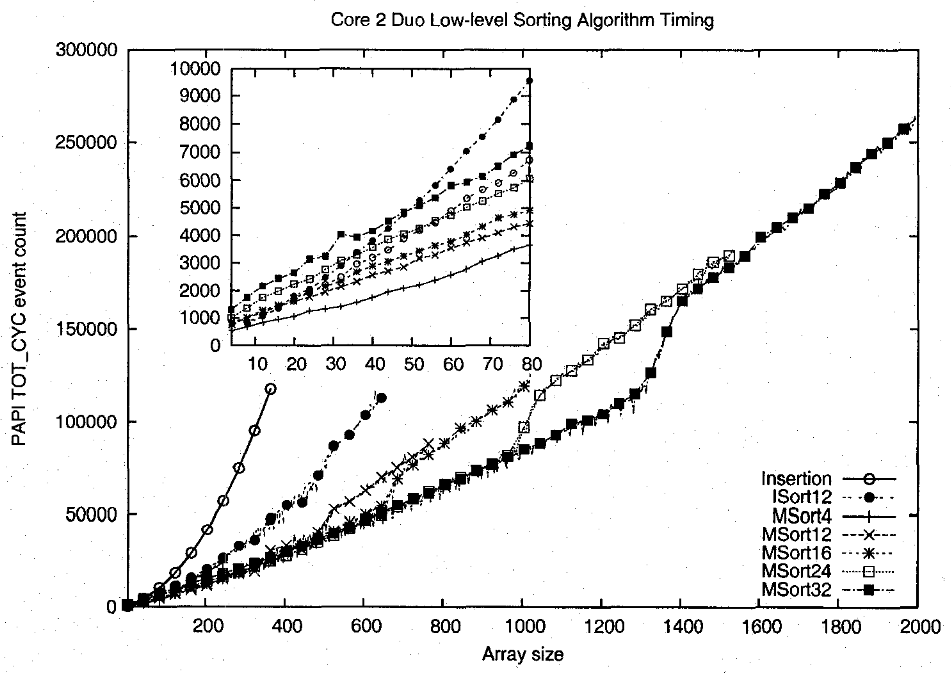


Figure 4.2: Low-level algorithm cycle counts for MSort on the Core 2 Duo.

2M of the Core 2 Duo and the Pentium 4.

The low-level performance for moderate arrays on the G5 closely follows that of the Athlon 64, with the exception that ISort performs somewhat worse. For small arrays insertion sort performs surprisingly well, keeping pace with ISort and MSort until about 100 elements.

Pentium 4

Following the trend of varying execution profiles, the Pentium 4 exhibits essentially straightline performance for all versions of MSort up to its full range of 2048 elements as seen in Fig 4.6. For small arrays RSort is easily the winner, followed by MSort, and then ISort.

4.4 Quicksort

4.4.1 Methodology

To measure Quicksort performance we integrated our collection of low-level scalar and SIMD-based sorting algorithms into the non-recursive Quicksort implementation described in Chapter 3. When the number of elements to be sorted drops below a threshold the default Quicksort switches to a scalar sorting network. For each algorithm's threshold on each architecture we use the empirical value given in Appendix A. The reference partitioning function used is S1, the scalar binary partition. This version of Quicksort is the baseline for the graphs of the relative performance of each algorithm.

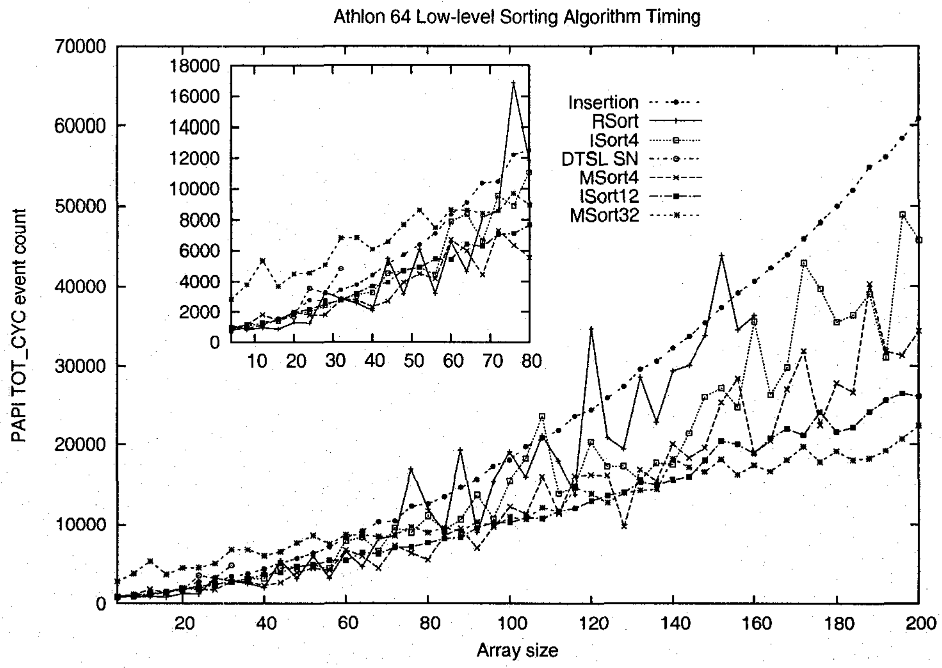


Figure 4.3: Low-level algorithm cycle counts on the Athlon 64.

Five different distributions were examined: (1) a uniform distribution with a range much larger than the array; (2) a uniform distribution with a range of 100; (3) strictly increasing; (4) strictly decreasing; and (5) a "pipe organ" distribution, with the smallest values at the ends of the array, increasing towards the middle.

We tested array sizes of 200, 500, 2k, 5k, 10k, 20k, 40k, and 80k using 2000 sample runs, after 5 warm-up runs. Arrays of 1M and 10M elements were tested using 20 samples and 5 warm-up runs.

All SIMD sorting algorithms are called from within a wrapper function to correct for misaligned sub-arrays. As noted in Section 3.5, this wrapping should not significantly affect performance.

4.4.2 Results

Core 2 Duo

Fig. B.1 shows the common case of uniformly distributed data. Looking at the scalar partitioning algorithms S1 and S2, when the array size increases the contribution of the low-level sorting algorithms, as expected, begins to be surpassed by the partitioning effort. For the smallest arrays (200 elements), which are small enough to be handled immediately by MSort, execution time, compared to the baseline, may be reduced 40-50%. For 10M elements this reduction drops to around 12% when partitioning with S1. S2 involves greater partitioning overhead and this is reflected in its comparatively worse times.

Using vector partitioning functions V1 and V2 shows a clear increase in performance over the scalar versions, with the best MSort and RSort time reductions near 30%, even for 10M elements.

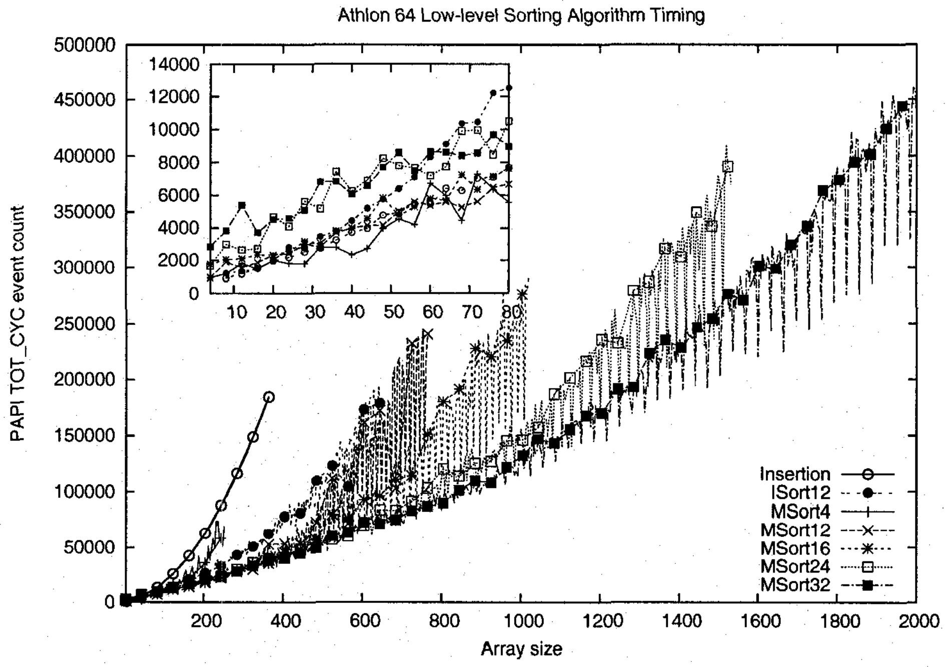


Figure 4.4: Low-level algorithm cycle counts for MSort on the Athlon 64.

Neither the two uniform nor the pipe-organ ordistribution are suited to show a significant difference between V1 and V2. In the case of increasing keys in Fig. B.3, V2 shows a clear advantage over V1. V1's performance steadily drops, going below the baseline as the array size increases. Comparatively V2 is able to keep RSort and the best MSort versions at or near a 10% gain for 10M elements. V2's performance for the decreasing distribution in Fig. B.4 is somewhat more ragged, but still positive.

As intended, V2 is able to sufficiently exploit regularities in the array being partitioned while maintaining the speed of V1 for uniformly distributed keys.

G5

The good performance of V1 and V2 is not replicated on the G5. Indeed the vector partitioning functions are consistently bad on all but the first distribution (uniform values with a large variance). This drop in performance can be attributed to the extra effort needed to compute the 4-bit comparison value from the comparison mask, compared to the single instruction on the x86-64.

The variants of MSort are able to perform reasonably well using the scalar binary and ternary partitioning functions S1 and S2, while ISort is consistently worse than MSort, and often the baseline as well.

Pentium 4 and Athlon 64

The Pentium 4 and Athlon 64 both loosely follow the trends of the Core 2 Duo — not particularly surprising given their shared instruction set. Although performance for the Athlon 64 is good on the

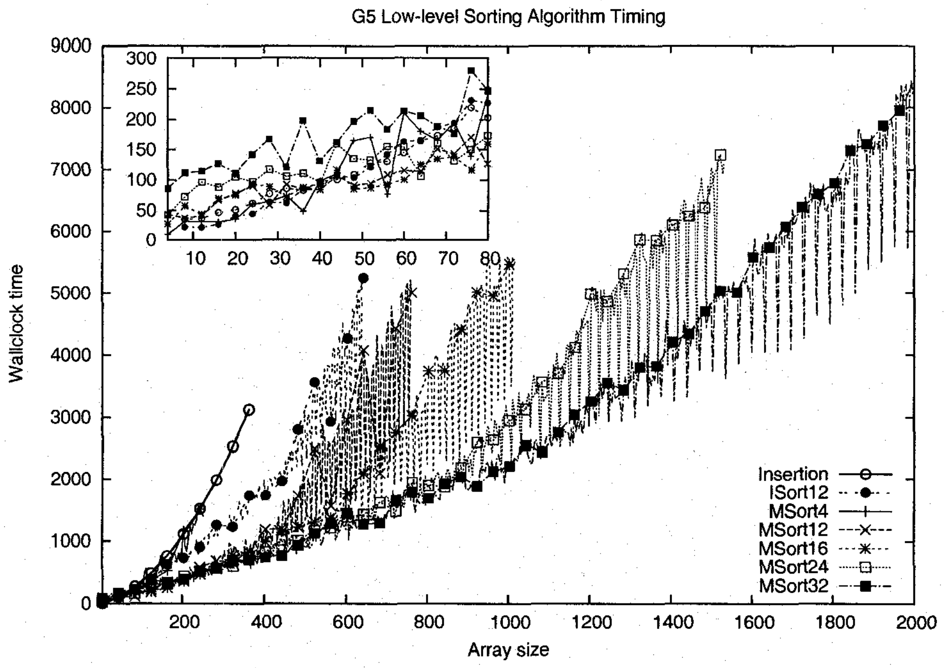


Figure 4.5: Low-level algorithm cycle counts for MSort on the G5.

large uniform distribution, it tends to perform notably worse than the other two on the final three distributions.

4.5 Heapsort

4.5.1 Methodology

Using the *d*-heap operations defined in Chapter 2.5, we implemented scalar and SIMD versions of heapsort. The *d* value for the sorting heaps was known at compile time. For the SIMD variants, we implemented the described heapsort wrapper function to correct for the case where the first child node is unaligned — a linear scan locates the smallest element and moves it to the first position, effectively shifting the unsorted array. We investigated all four combinations of using or omitting Floyd's variants for both *sift-down* and for the initial *make-heap* phase. Recall that these variations reduce the number of instructions at the expense of poorer data locality.

We used these Heapsort versions to sort not-necessarily-aligned arrays of sizes in powers of 2, from 2^2 to 2^{22}. Heapsort applied to arrays with 2^{22} elements was sampled 5 times. For each decrease in the exponent one addition sample run was added, as smaller heaps are prone to more measurement noise, up to 29 samples for 2^2 elements. In each case one additional run was first discarded to load the relevant data into the processor's cache.

We present results for sorting uniformly distributed keys with a large range. As with the previous experiments, heap elements are key-pointer pairs.

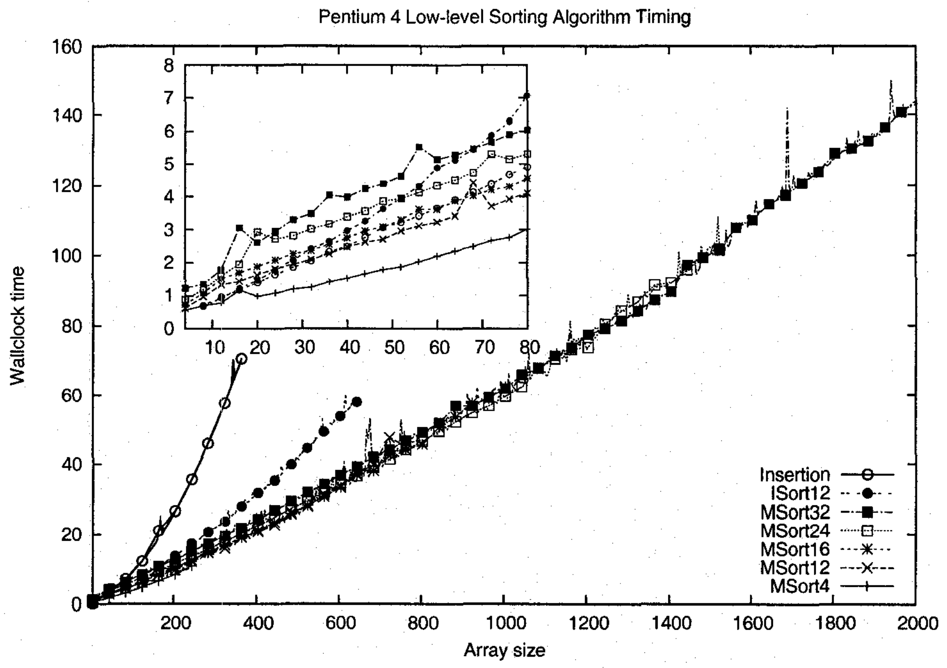


Figure 4.6: Low-level algorithm cycle counts for MSort on the Pentium 4.

4.5.2 Results

To distinguish between the various flavours of each *d* value, the tables in Appendix C use the following suffixes: "F" to denote Floyd's make-heap variant was used, rather than the incremental-add method; "V" to denote the SIMD variant which uses vector instructions to find the max child node. Floyd's sift-down variant was not found to be particularly helpful and is not included.

Core 2 Duo

Up until 2^{12} elements the standard binary heap with Floyd's modification is the best performing on the Core 2 Duo. Beyond that point the title shifts between branching factors of 12, 16, and 24, with and without Floyd's modification. The performance of these heaps are all comparable for these sizes, with gains of approximately 17% relative to the non-vector versions at the upper end of the array sizes.

Pentium 4

The Pentium 4 shows a substantive shift to the vectorized *d*-heap, using Floyd's variant, once the array size reaches approximately 2^{12}. From that point the 16-way heap is faster than or equal to all other variants.

Athlon 64

In the case of the Athlon 64, similar to the Core 2 Duo, the binary heap with Floyd's make-heap modifications is dominant up until the array size reaches 2^{16}, at which point the 12-way vectorized heap becomes the best. This crossover point coincides with the L2 cache size on the Athlon 64, and reiterates LaMarca's observation that increased fanouts are superior once the heap grows beyond the processor's cache.

G5

Results for the G5 are consistently in favour of the SIMD heap operations, with reductions of around 20% for the larger heaps.

4.6 Summary

We have demonstrated that the combination of low-level SIMD sorting algorithms integrated into quicksort, combined with a vectorized partitioning function can result in significant speed improvements over a highly efficient baseline implementation. The Core 2 Duo saw a time reduction of approximately 30% for uniformly distributed data. Execution times on the G5 were reduced by 10-20% using standard binary partitioning, with the SIMD partitioning performing terribly.

Using some flavour of MSort within quicksort is almost always a strong improvement – moreso if the distribution is known to be uniform. The extent to which using these low-level algorithms resulted in performance gains was limited on those machines with small amounts of cache, although other architectural factors may have contributed.

For sorting short sequences of data, RSort has the potential to be almost twice as fast as other algorithms, and many times faster than insertion sort. This may be a great benefit in situations calling for sorting many short arrays.

For all architectures the best heapsort performance for large arrays was obtained using SIMD instructions. The crossover point at which SIMD heap operations become practical may be too high for its use as a fallback sort within introsort, except perhaps on the G5 and the Pentium 4. This technique may still be successfully employed in situations where worst-case performance must be guaranteed, such that heapsort is always used to sort the entire array.

Chapter 5

Conclusions and Future Work

This thesis proposes the use of the SIMD machinery provided in modern processors to improve the performance of recursion tails. The idea is that whenever the number of elements to be processed fits within the SIMD registers available in the processor, these values should be loaded once into the SIMD registers and then an efficient SIMD execution should be used. While the feasibility of this idea was demonstrated with the integration into quicksort of a more efficient algorithm for sorting short sequences, the idea should be generally applicable to recursive computation.

Once efficient low-level SIMD algorithms are crafted, they can be generated into a solution database to be instantiated by code generators into optimized libraries. Alternatively, if a suitable identification algorithm is created, the compiler should be able to integrate these solutions directly into general programs.

The main results of this thesis are:

- MSort is a fast and robust sorting algorithm that can handle moderately sized arrays well and has few parameters to adjust. Quicksort performance improves by 10-30% over baseline on large arrays with a uniform key distribution, when combined with vector partitioning.
- RSort is excellent for sorting short sequences of elements – approximately 2.5 times faster than insertion sort for fewer than 64 elements on the Core 2 Duo.
- Cache sizes may limit the performance of algorithms such as SSort and RSort that are written as many sequential instructions. Larger caches allow for more effective sorting in these cases.
- Vectorized partitioning functions can be highly beneficial, with additional time reduction of up to 20% over MSort on uniformly distributed data, and exploiting common input distributions can be done without notable overhead.

5.1 Future Work

Although we have generated efficient instruction sequences for aligning data and executing multiple comparators in parallel for RSort, we do not claim that these sequences are optimal. We do not feel it likely that *large* gains could be made by investing additional search effort, but small gains are certainly possible and may warrant investigation if execution time is critical. This would most likely be the case for situations involving sorting many short sequences, where RSort is superior, rather than divide-and-conquer scenarios where such benefit would be diluted.

Additionally, it may be the case that there is a deterministic sorting network (not necessarily scalar-optimal), which lends itself to efficient alignment instruction sequences. Such a network may operate along the lines of the perfect shuffling sorting network algorithms used in vector processors.

The vectorized partitioning function was implemented with a first pass to detect data at both ends of the array that was already properly located. It should be possible to include this functionality, so as to avoid extraneous write instructions, inside the main loop at the expense of some additional bookkeeping. This bookkeeping may be encoded in the program state.

Our experimental study focused on array sizes that were small enough to fit within main memory. This is often not the case for large databases, where records must be read to and from disk. It would be worthwhile to determine the extent to which such SIMD optimizations might affect performance on very large arrays, where disk I/O latency becomes an important factor.

Bibliography

[1] M. Ajtai, J. Komlós, and E. Szemerédi. An O(n log n) sorting network. In *STOC '83: Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 1–9, New York, NY, USA, 1983. ACM Press.

[2] K. E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, pages 307–314, 1968.

[3] J. L. Bentley and M. D. McIlroy. Engineering a sort function. *Softw. Pract. Exper.*, 23(11):1249–1265, 1993.

[4] L. Bishop, D. Eberly, T. Whitted, M. Finch, and M. Shantz. Designing a PC game engine. *IEEE Computer Graphics and Applications*, 18(1):46–53, 1998.

[5] D. Bitton, D. J. DeWitt, D. K. Hsiao, and J. Menon. A taxonomy of parallel sorting. *Computing Surveys*, 16(3):287–318, September 1984.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition.* MIT Press, Cambridge, MA, USA, 2001.

[7] E. W. Dijkstra. *Executional abstraction*, chapter 14. Prentice-Hall, 1976.

[8] J. D. Frens and D. S. Wise. Auto-blocking matrix-multiplication or tracking BLAS3 performance from source code. In *Principples and Practice of Parallel Programming PPoPP*, pages 206–216, Las Vegas, Nevada, 1997.

[9] M. Frigo. A fast Fourier transform compiler. In *Programming Language Design and Implementation PLDI*, pages 169–180, Atlanta, GA, June 1999.

[10] T. Furtak. C++ source code for a vectorized sorting library. http://www.cs.ualberta.ca/ furtak/sort, 2007.

[11] T. Furtak, J. N. Amaral, and R. Niewiadomski. Using SIMD registers and instructions to enable instruction-level parallelism in sorting algorithms. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 348–357, New York, NY, USA, 2007. ACM Press.

[12] N. K. Govindaraju, N. Raghuvanshi, and D. Manocha. Fast and approximate stream mining of quantiles and frequencies using graphics processors. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 611–622, New York, NY, USA, 2005. ACM Press.

[13] Y. Han and Y. Igaraski. Time lower bounds for sorting on multi-dimensional mesh-connected processor arrays. In *Proceedings of the 1988 International Conference on Parallel Processing*, volume III, Algorithms and Applications, pages 194–197, University Park, Penn, 1988. Penn State.

[14] C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4(7):321, 1961.

[15] Intel. IA-32 Intel®64 and ai-32 architectures software developer's manual volume 1: Basic architecture. http://www.intel.com/design/processor/manuals/253665.pdf, 2007.

[16] M. Kik, M. Kutylowski, and G. Stachowiak. Periodic constant depth sorting networks. In *Symposium on Theoretical Aspects of Computer Science*, pages 201–212, 1994.

[17] K. C. Kiwiel. Partitioning schemes for quicksort and quickselect. *ArXiv Computer Science e-prints*, December 2003.

[18] D. E. Knuth. *The Art of Computer Programming, Vol. 3 - Sorting and Searching*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1973.

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

[20] A. LaMarca and R. E. Ladner. The influence of caches on the performance of sorting. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1997.

[21] X. Li, M. Garzaran, and D. Padua. A dynamically tuned sorting library. In *Code Generation and Optimization CGO*, pages 111–122, Palo Alto, CA, March 2004.

[22] X. Li, M. J. Garzarán, and D. Padua. Optimizing sorting with genetic algorithms. In *Code Generation and Optimization CGO*, pages 99–110, San Jose, CA, March 2005.

[23] D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for SIMD. In *Programming language design and implementation PLDI*, pages 132–143, 2006.

[24] Stephan Olariu, M. Christina Pinotti, and S. Q. Zheng. How to sort n items using a sorting network of fixed i/o size. *IEEE Trans. Parallel Distrib. Syst.*, 10(5):487–499, 1999.

[25] I. Parberry. On the computational complexity of optimal sorting network verification. In *PARLE (1)*, pages 252–269, 1991.

[26] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 1992.

[27] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 41–50. Eurographics Association, 2003.

[28] A. Ranade, S. Kothari, and R. Udupa. Register efficient mergesorting. In *High Performance Computing — HiPC*, volume 1970 of *LNCS*, pages 96–103. Springer, 2000.

[29] G. Ren, P. Wu, and D. Padua. Optimizing data permutations for SIMD devices. In *Programming language design and implementation PLDI*, pages 118–131, 2006.

[30] X. Shen and C. Ding. Adaptive data partition for sorting using probability distribution. In *ICPP '04: Proceedings of the 2004 International Conference on Parallel Processing (ICPP'04)*, pages 250–257, Washington, DC, USA, 2004. IEEE Computer Society.

[31] H. J. Siegel. The universality of various types of SIMD machine interconnection networks. In *Proceedings of the 4th Annual Symposium on Computer Architecture*, pages 23–25, Silver Spring, MD, March 1977. ACM SIGARCH/IEEE-CS.

[32] C. D. Thompson and H. T. Kung. Sorting on a mesh-connected parallel computer. In *STOC '76: Proceedings of the eighth annual ACM symposium on Theory of computing*, pages 58–64, New York, NY, USA, 1976. ACM Press.

[33] S. A. A. Touati. Register saturation in instruction level parallelism. *International Journal of Parallel Programming*, 33(4):393–449, 2005.

[34] R. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of sotware and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.

[35] R. Wickremesinghe, L. Arge, J. S. Chase, and J. S. Vitter. Efficient sorting using registers and caches. *ACM Journal of Experimental Algorithmics*, 7:9, 2002.

[36] J. Xiong, J. Johnson, R. Johnson, and D. Padua. SPL: A language and compiler for DSP algorithms. In *Programming Language Design and Implementation PLDI*, pages 298–308, Snowbird, Utah, June 2001.

Appendix A

Quicksort Thresholds

Best Quicksort Thresholds for the Core 2 Duo

Algorithm		Array Size					Avg. Thresh.
		5000	10000	20000	40000	80000	
MSort4	*S1*	100	100	100	100	104	**100**
	V1	88	84	80	100	84	**84**
	S2	104	104	80	80	84	**88**
MSort8	*S1*	136	196	168	132	136	**152**
	V1	132	132	132	132	132	**132**
	S2	228	200	224	208	176	**204**
MSort12	*S1*	264	288	240	276	240	**260**
	V1	192	192	216	196	212	**200**
	S2	292	288	288	276	284	**284**
MSort16	*S1*	512	512	512	512	384	**484**
	V1	260	388	332	416	364	**352**
	S2	568	560	512	552	512	**540**
MSort20	*S1*	532	480	480	476	480	**488**
	V1	480	484	480	500	336	**456**
	S2	484	548	480	480	544	**504**
MSort24	*S1*	576	576	600	604	612	**592**
	V1	576	576	576	544	432	**540**
	S2	576	584	620	616	580	**592**
MSort28	*S1*	616	896	616	896	560	**716**
	V1	592	452	560	604	616	**564**
	S2	1116	1044	1112	964	960	**1036**
MSort32	*S1*	980	960	1024	1024	1024	**1000**
	V1	1024	1024	1024	1024	768	**972**
	S2	1024	1212	1144	1084	1084	**1108**
ISort4	*S1*	100	100	100	100	100	**100**
	V1	68	68	56	68	68	**64**
	S2	80	80	100	92	80	**84**
ISort8	*S1*	120	132	132	132	132	**128**
	V1	132	132	132	96	140	**124**
	S2	160	164	164	180	156	**164**
ISort12	*S1*	196	196	172	212	200	**192**
	V1	200	196	188	176	188	**188**
	S2	192	200	192	188	188	**192**
RSort	*S1*	40	40	40	40	40	**40**
	V1	40	40	40	40	40	**40**
	S2	44	40	40	44	40	**40**
DTSL	*S1*	20	20	20	20	20	**20**
	V1	20	20	20	20	20	**20**
	S2	20	24	24	24	20	**20**
Insertion	*S1*	56	64	56	88	80	**68**
	V1	96	64	104	76	92	**84**
	S2	128	116	116	80	120	**112**

Table A.1: Quicksort thresholds for Core 2 Duo (overclocked).

Best Quicksort Thresholds for the Pentium 4

Algorithm		Array Size					Avg. Thresh.
		5000	10000	20000	40000	80000	
MSort4	*S1*	128	128	124	124	140	**128**
	V1	124	120	124	104	124	**116**
	S2	216	180	148	148	152	**168**
MSort8	*S1*	192	256	256	256	256	**240**
	V1	256	256	232	288	256	**256**
	S2	304	308	292	308	296	**300**
MSort12	*S1*	288	288	288	288	288	**288**
	V1	536	384	288	288	308	**360**
	S2	580	580	512	500	516	**536**
MSort16	*S1*	512	512	512	512	512	**512**
	V1	512	512	512	512	512	**512**
	S2	612	780	604	560	520	**612**
MSort20	*S1*	828	764	688	480	552	**660**
	V1	952	480	480	480	480	**572**
	S2	948	984	880	896	828	**904**
MSort24	*S1*	1032	576	768	576	588	**708**
	V1	1084	860	600	576	576	**736**
	S2	1060	1108	1140	968	984	**1052**
MSort28	*S1*	896	896	896	896	896	**896**
	V1	1204	1208	896	896	896	**1020**
	S2	1204	1100	1192	892	1104	**1096**
MSort32	*S1*	1024	1024	1024	1024	1024	**1024**
	V1	1024	1024	768	1024	1024	**972**
	S2	1628	1512	1448	1112	1052	**1348**
ISort4	*S1*	128	128	124	120	112	**120**
	V1	96	112	120	100	112	**108**
	S2	144	152	152	128	124	**140**
ISort8	*S1*	124	132	192	144	132	**144**
	V1	128	120	120	124	132	**124**
	S2	248	284	268	272	272	**268**
ISort12	*S1*	196	196	196	196	196	**196**
	V1	172	196	196	192	188	**188**
	S2	308	304	304	292	288	**296**
RSort	*S1*	68	68	68	68	68	**68**
	V1	68	68	68	72	64	**68**
	S2	120	124	112	116	100	**112**
DTSL	*S1*	12	12	12	12	12	**12**
	V1	12	12	12	12	12	**12**
	S2	20	20	20	20	20	**20**
Insertion	*S1*	128	128	128	60	60	**100**
	V1	80	128	76	128	128	**108**
	S2	80	88	80	84	80	**80**

Table A.2: Quicksort thresholds for Pentium 4.

Best Quicksort Thresholds for the Athlon 64

Algorithm		**Array Size**					**Avg. Thresh.**
		5000	10000	20000	40000	80000	
MSort4	*S1*	76	76	80	76	80	**76**
	V1	84	68	68	80	80	**76**
	S2	80	80	84	80	80	**80**
MSort8	*S1*	148	192	132	140	132	**148**
	V1	196	196	132	132	132	**156**
	S2	164	160	160	156	160	**160**
MSort12	*S1*	248	240	240	244	240	**240**
	V1	244	240	196	244	268	**236**
	S2	288	288	248	248	248	**264**
MSort16	*S1*	484	328	320	272	260	**332**
	V1	328	328	260	260	388	**312**
	S2	444	404	396	364	364	**392**
MSort20	*S1*	576	480	412	464	352	**456**
	V1	404	448	360	328	404	**388**
	S2	596	536	372	496	420	**484**
MSort24	*S1*	576	572	480	480	428	**504**
	V1	576	484	484	388	408	**468**
	S2	732	664	512	452	464	**564**
MSort28	*S1*	672	672	560	452	452	**560**
	V1	452	452	452	456	452	**452**
	S2	940	776	576	656	624	**712**
MSort32	*S1*	908	772	516	708	772	**732**
	V1	772	776	516	516	648	**644**
	S2	1016	764	680	676	716	**768**
ISort4	*S1*	84	68	76	76	76	**76**
	V1	68	68	80	68	80	**72**
	S2	80	80	84	84	88	**80**
ISort8	*S1*	132	132	128	128	124	**128**
	V1	132	132	132	132	132	**132**
	S2	156	156	156	156	144	**152**
ISort12	*S1*	196	196	196	200	196	**196**
	V1	196	196	180	176	196	**188**
	S2	228	188	204	180	188	**196**
RSort	*S1*	52	52	52	52	52	**52**
	V1	52	52	52	44	52	**48**
	S2	56	56	52	52	52	**52**
DTSL	*S1*	20	20	20	20	20	**20**
	V1	20	20	20	20	20	**20**
	S2	24	24	20	24	20	**20**
Insertion	*S1*	32	32	32	48	48	**36**
	V1	48	48	48	60	128	**64**
	S2	80	80	80	80	80	**80**

Table A.3: Quicksort thresholds for Athlon 64.

Best Quicksort Thresholds for the G5

Algorithm		**Array Size**					**Avg.**
		5000	10000	20000	40000	80000	**Thresh.**
	S1	84	80	80	68	72	**76**
MSort4	*V1*	108	96	80	84	80	**88**
	S2	96	96	84	80	80	**84**
	S1	188	184	180	144	148	**168**
MSort8	*V1*	188	180	172	172	156	**172**
	S2	208	180	192	156	156	**176**
	S1	304	284	240	212	208	**248**
MSort12	*V1*	316	264	236	260	236	**260**
	S2	264	276	260	264	220	**256**
	S1	384	376	308	292	272	**324**
MSort16	*V1*	384	380	348	332	316	**352**
	S2	384	436	388	376	308	**376**
	S1	492	488	372	340	360	**408**
MSort20	*V1*	468	444	420	404	396	**424**
	S2	552	500	472	376	396	**456**
	S1	564	540	516	444	436	**500**
MSort24	*V1*	568	528	484	448	476	**500**
	S2	580	560	552	504	504	**540**
	S1	584	716	572	452	440	**552**
MSort28	*V1*	640	720	572	452	500	**576**
	S2	728	596	616	588	520	**608**
	S1	776	620	760	516	564	**644**
MSort32	*V1*	828	724	656	636	572	**680**
	S2	796	824	704	612	604	**708**
	S1	68	68	68	68	64	**64**
ISort4	*V1*	84	72	84	64	72	**72**
	S2	92	84	80	80	80	**80**
	S1	108	100	100	112	100	**104**
ISort8	*V1*	160	124	128	116	136	**132**
	S2	112	148	124	140	100	**124**
	S1	128	128	124	120	128	**124**
ISort12	*V1*	172	196	196	200	172	**184**
	S2	148	128	176	112	128	**136**
	S1	52	56	52	48	48	**48**
RSort	*V1*	56	52	52	48	52	**52**
	S2	68	60	60	52	52	**56**
	S1	28	32	28	32	28	**28**
DTSL	*V1*	32	32	32	28	32	**28**
	S2	32	32	32	32	28	**28**
	S1	52	52	48	48	48	**48**
Insertion	*V1*	80	60	60	80	60	**68**
	S2	40	44	44	44	80	**48**

Table A.4: Quicksort thresholds for G5.

Appendix B

Quicksort Timing

Core 2 Duo – Large Uniform Distribution

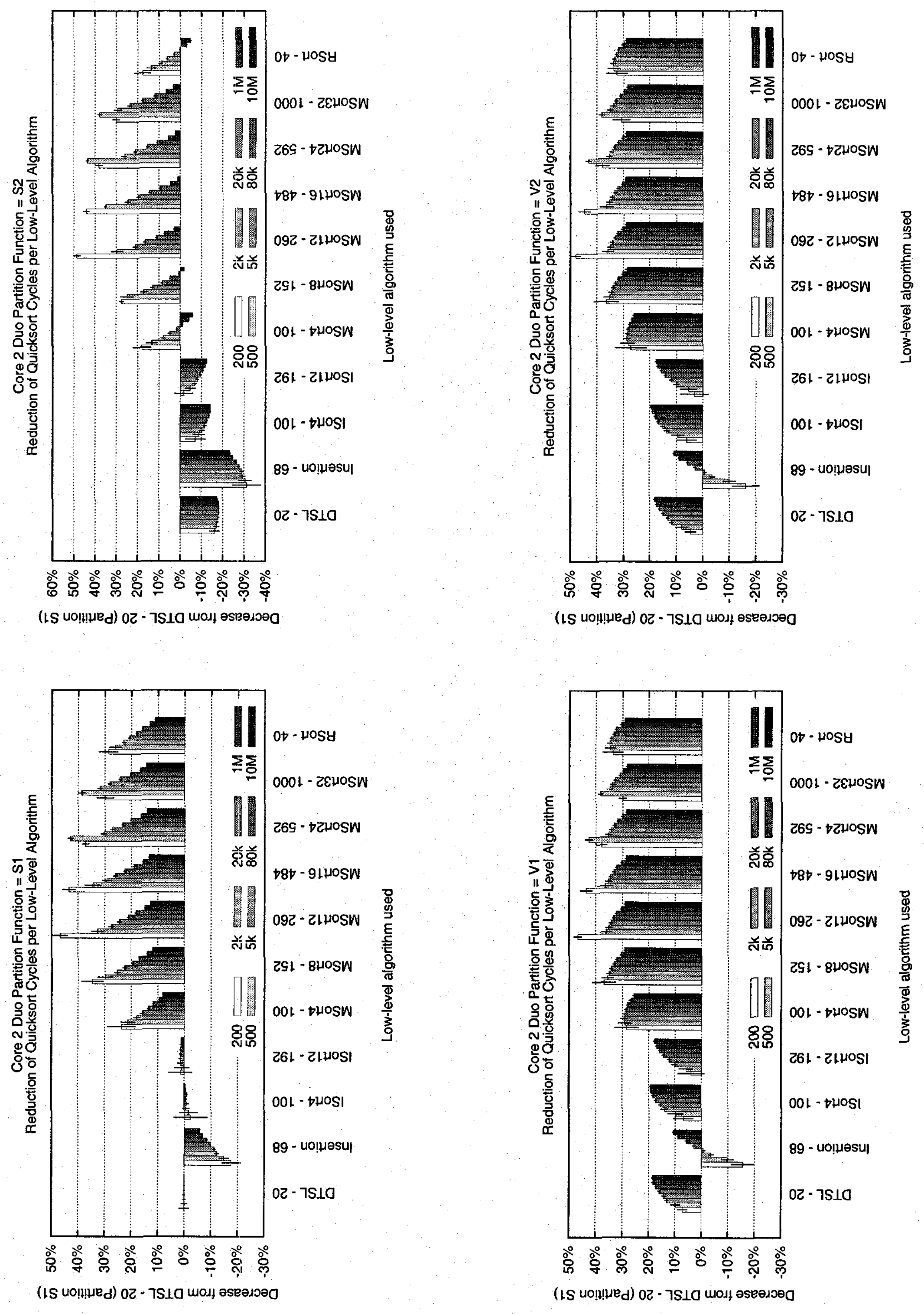


Figure B.1: Reduction in cycle counts relative to the baseline on an overclocked Core 2 Duo. Keys are drawn from a uniform distribution with a large range. Error bar is 1 standard deviation.

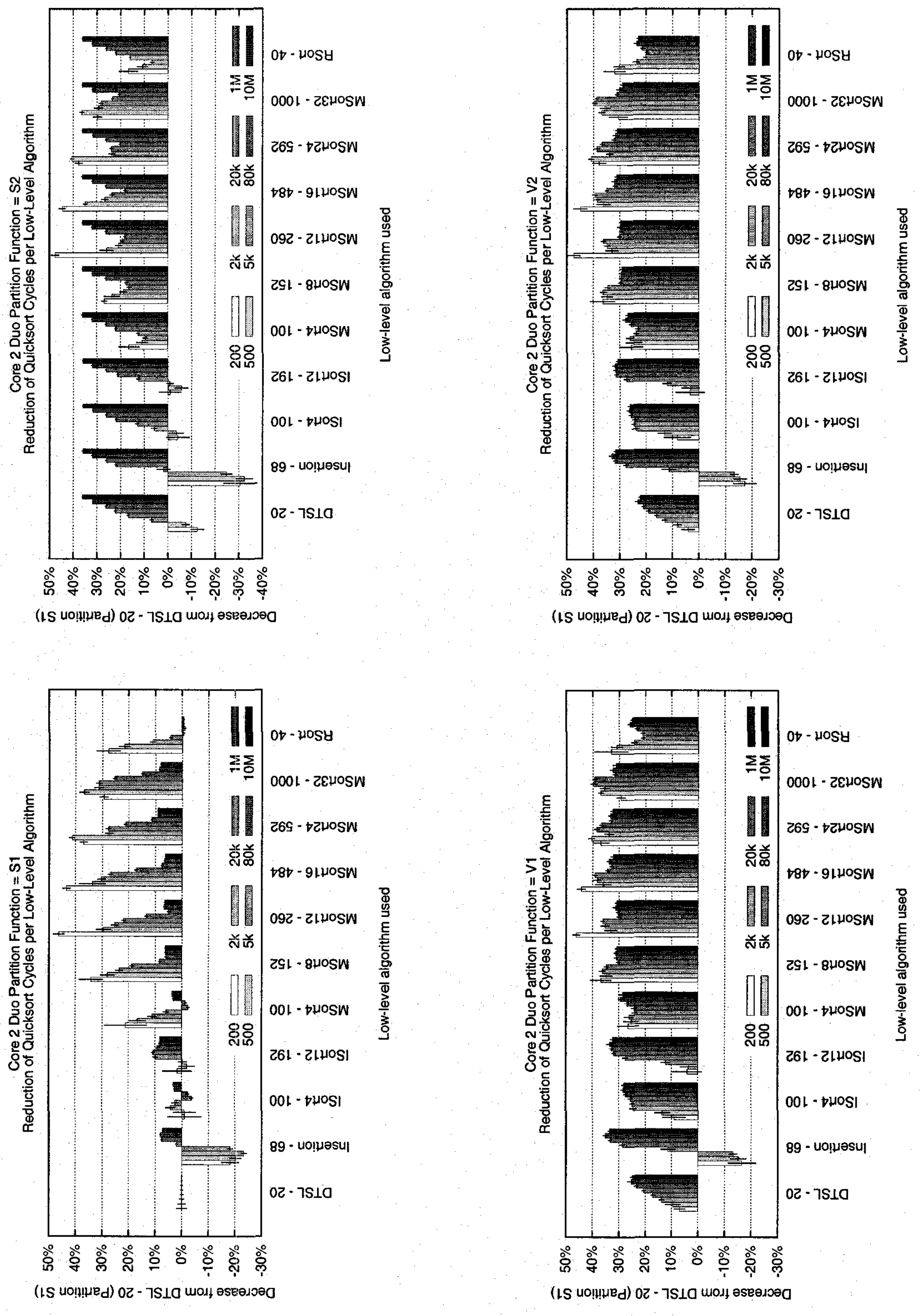
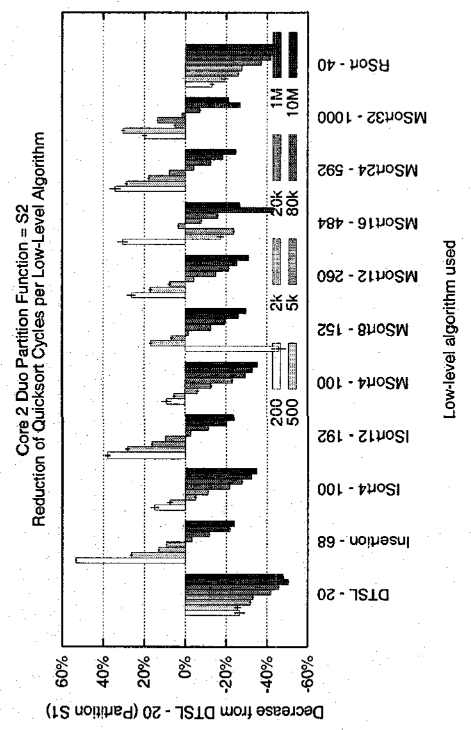
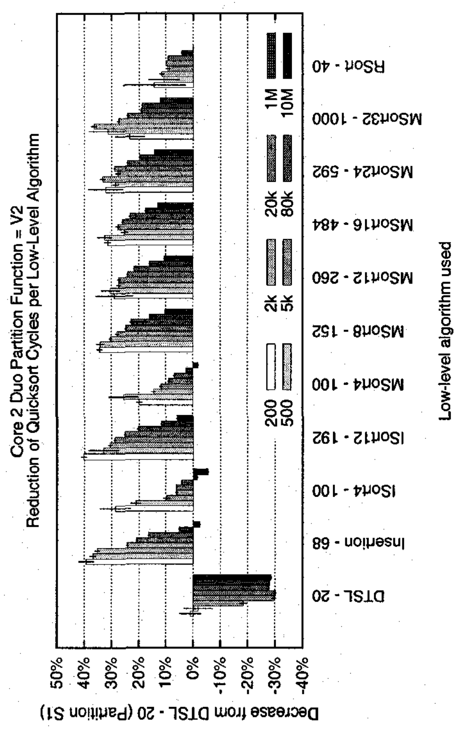
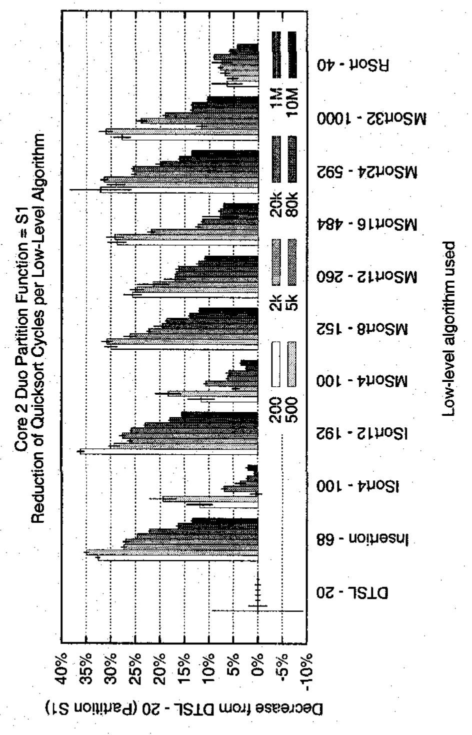


Figure B.2: Reduction in cycle counts relative to the baseline on an overclocked Core 2 Duo. Keys are drawn from a uniform distribution with a range of 100. Error bar is 1 standard deviation.







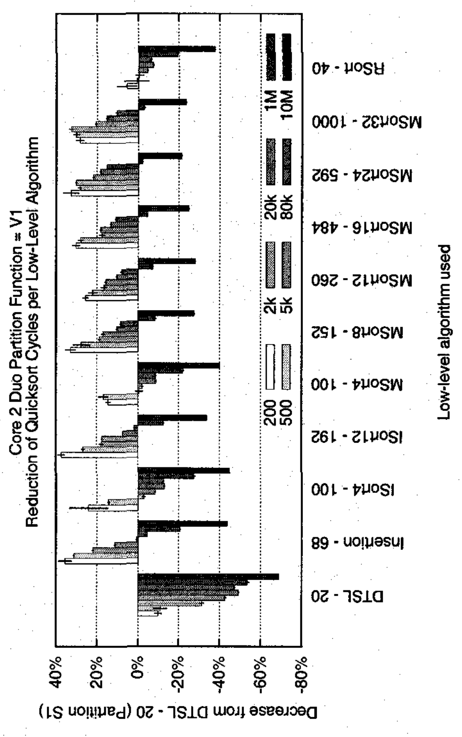
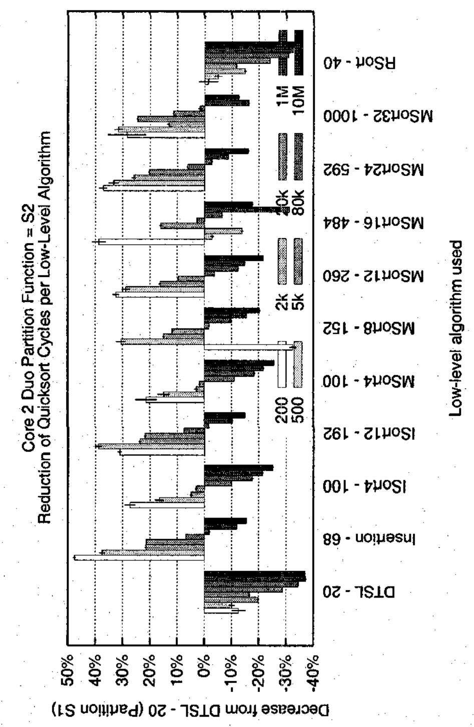
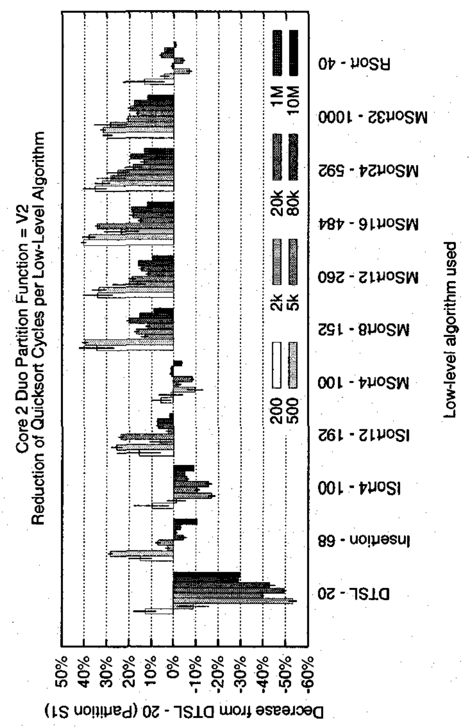
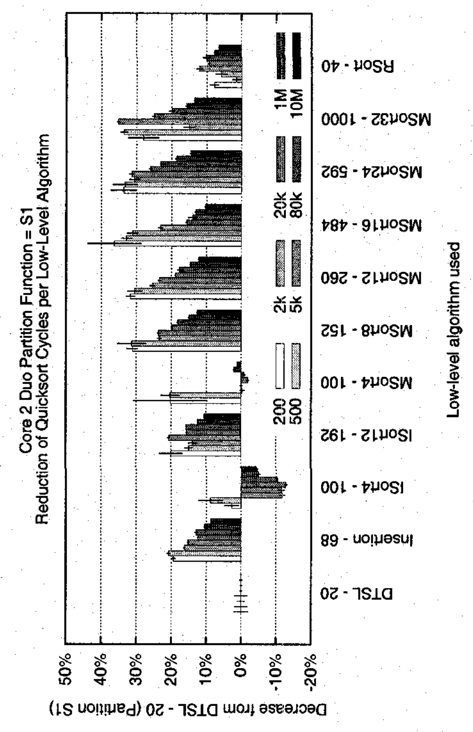


Figure B.3: Reduction in cycle counts relative to the baseline on an overclocked Core 2 Duo. Keys are initially in increasing order. Error bar is 1 standard deviation.







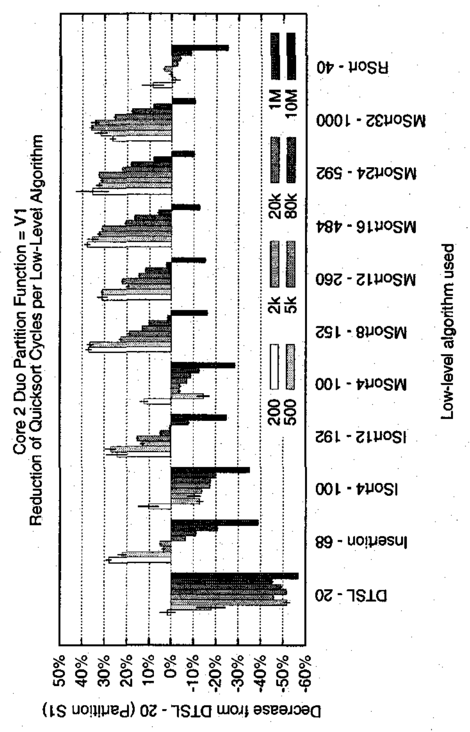


Figure B.4: Reduction in cycle counts relative to the baseline on an overclocked Core 2 Duo. Keys are initially in decreasing order. Error bar is 1 standard deviation.

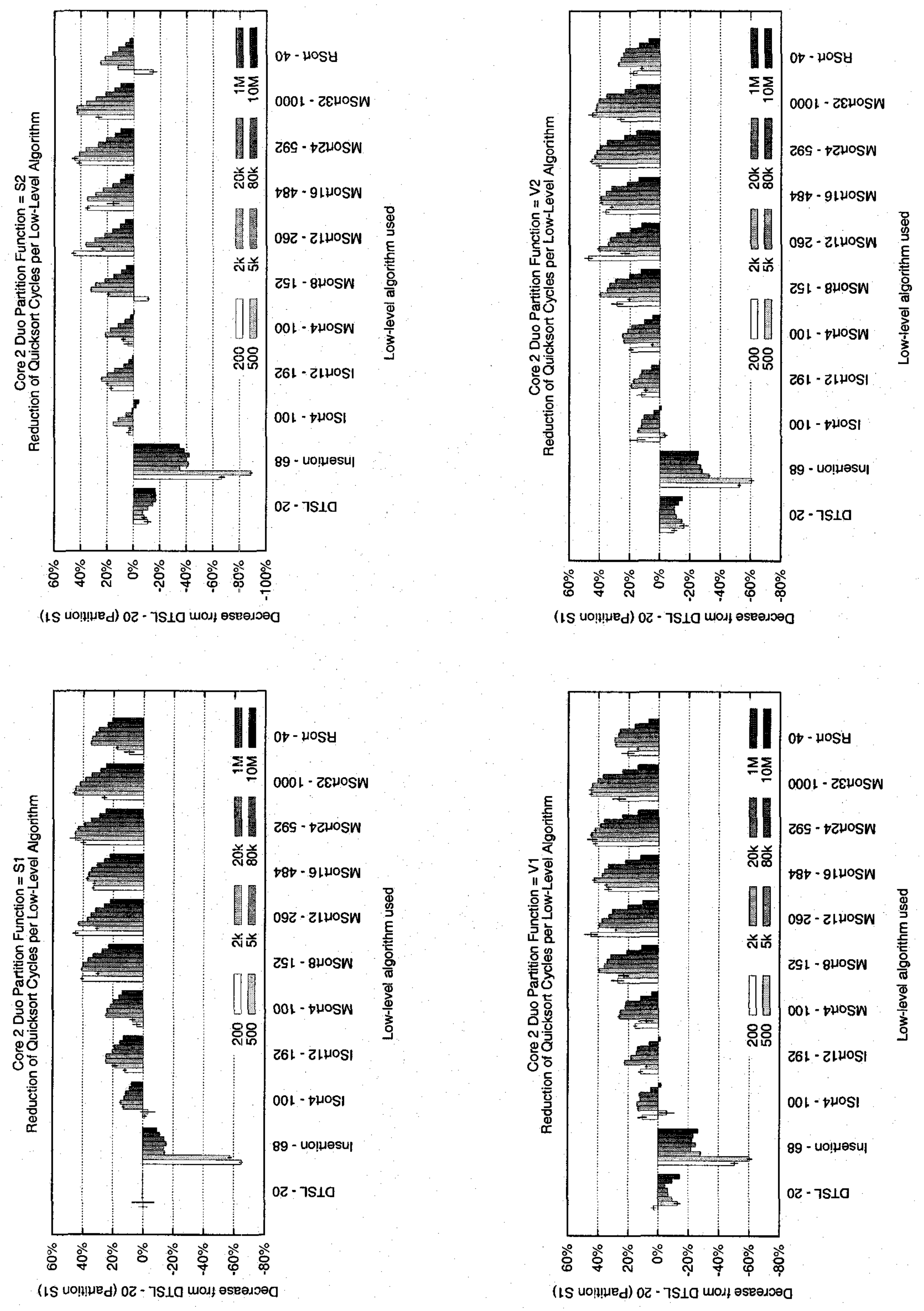
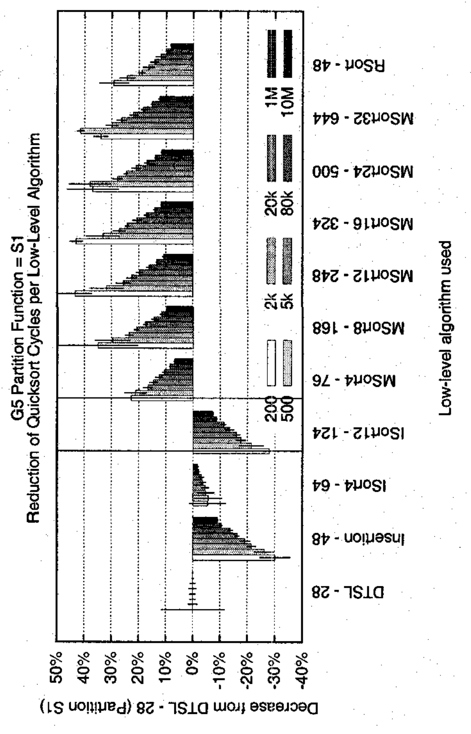
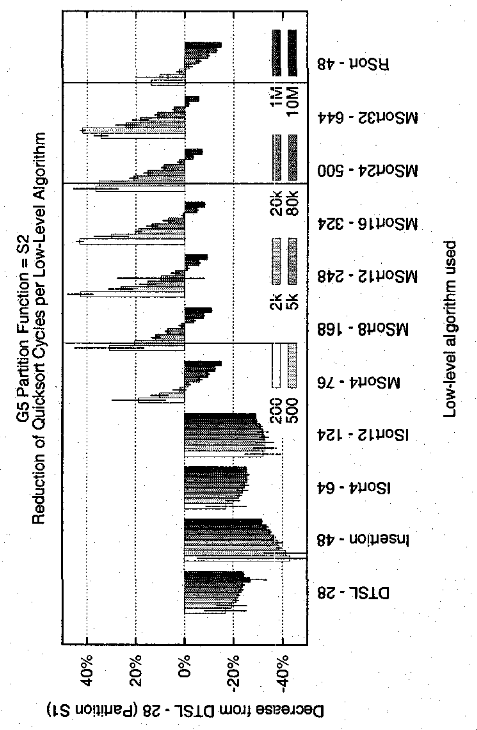
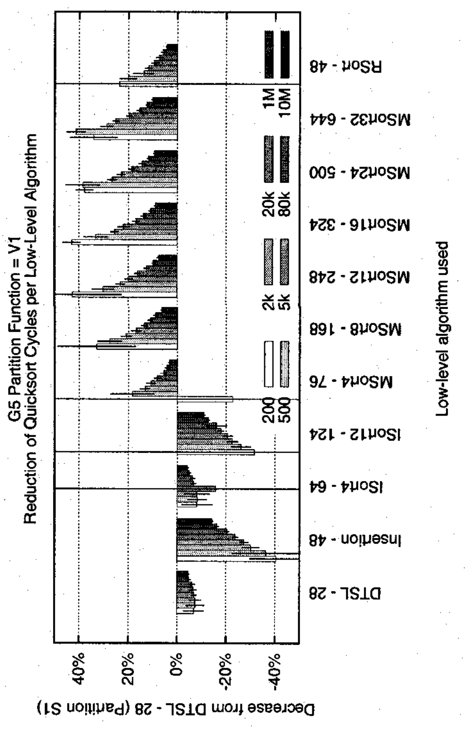


Figure B.5: Reduction in cycle counts relative to the baseline on an overclocked Core 2 Duo. Keys are initially increasing/decreasing with a peak at the middle. Error bar is 1 standard deviation.







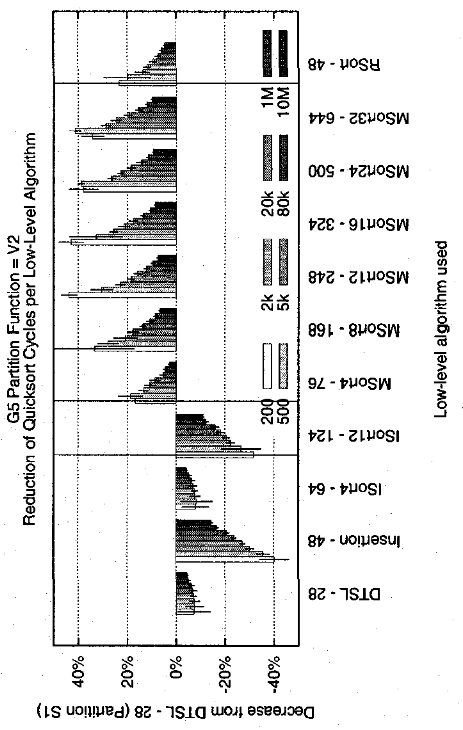


Figure B.6: Reduction in wall-clock time relative to the baseline on a G5. Keys are drawn from a uniform distribution with a large range. Error bar is 1 standard deviation.

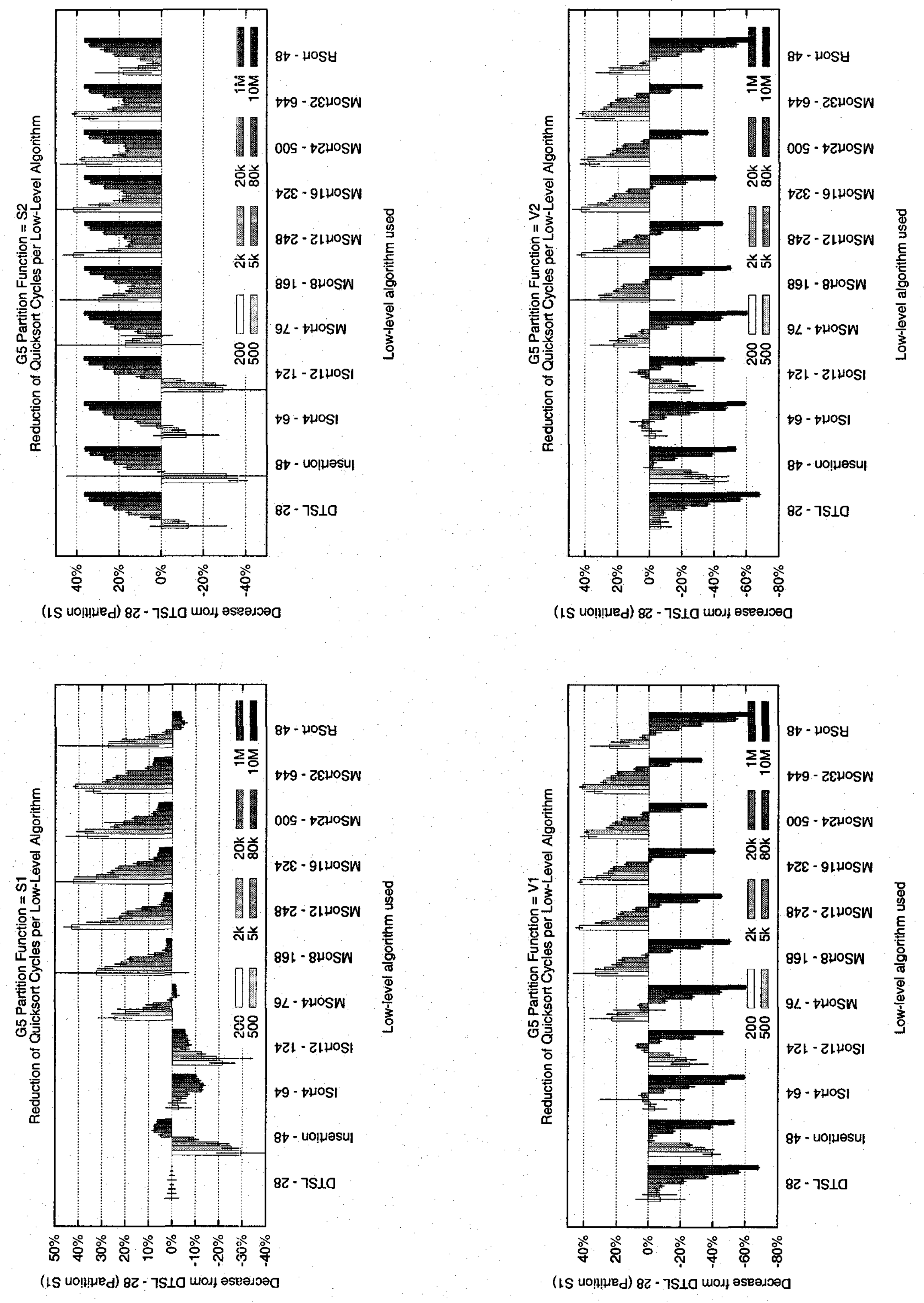
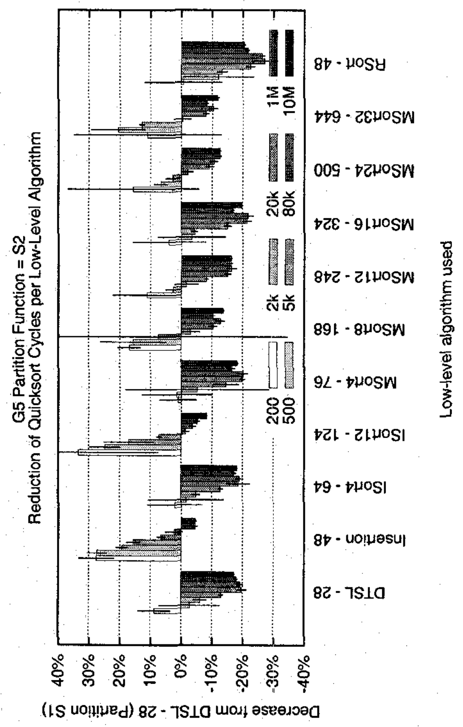
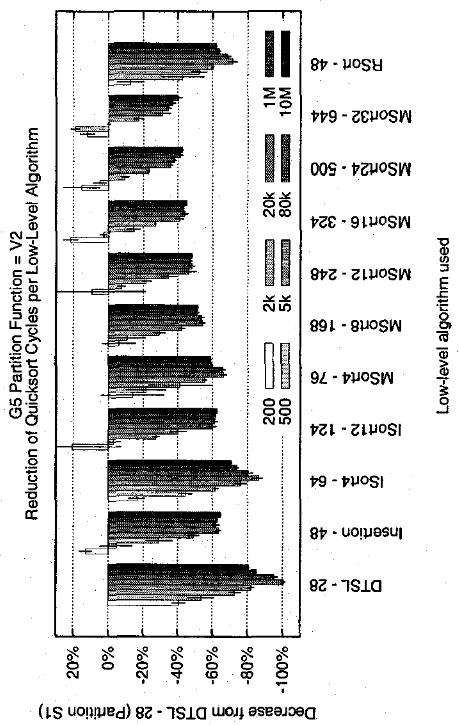
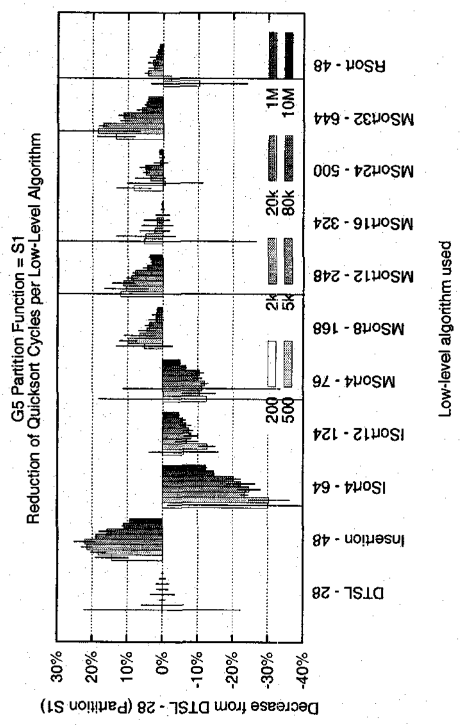


Figure B.7: Reduction in wall-clock time relative to the baseline on a G5. Keys are drawn from a uniform distribution with a range of 100. Error bar is 1 standard deviation.

G5 – Increasing Distribution







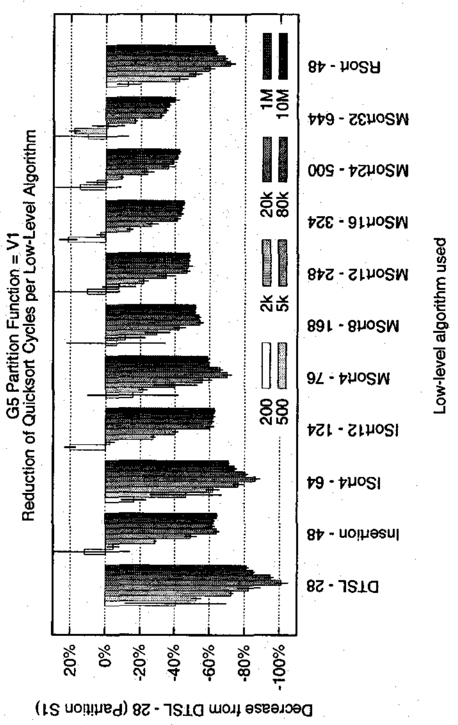
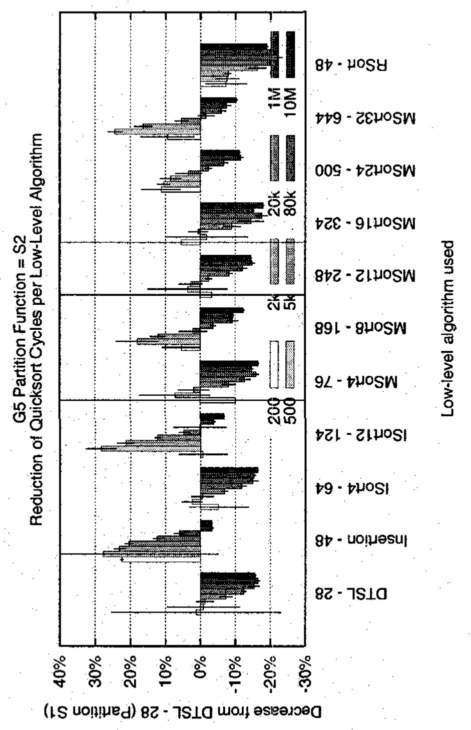
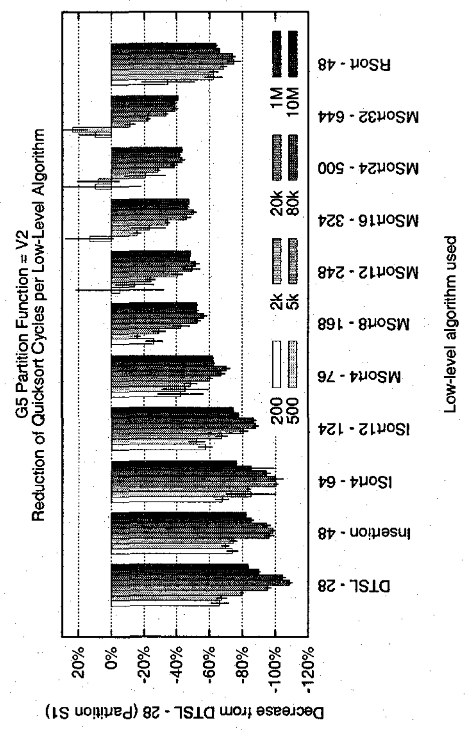
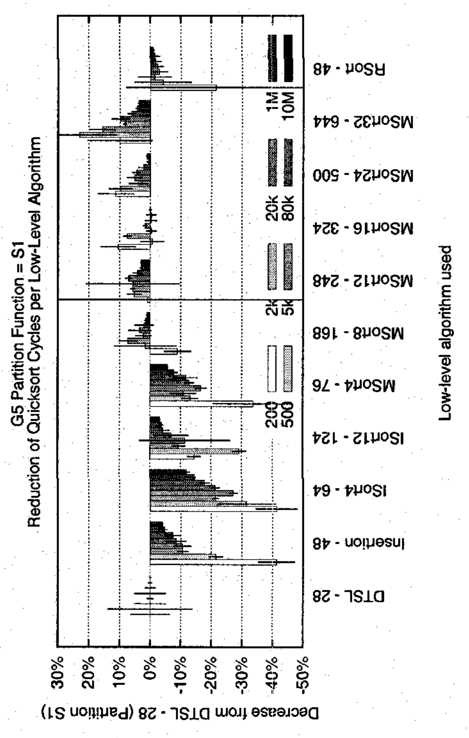


Figure B.8: Reduction in wall-clock time relative to the baseline on a G5. Keys are initially in increasing order. Error bar is 1 standard deviation.







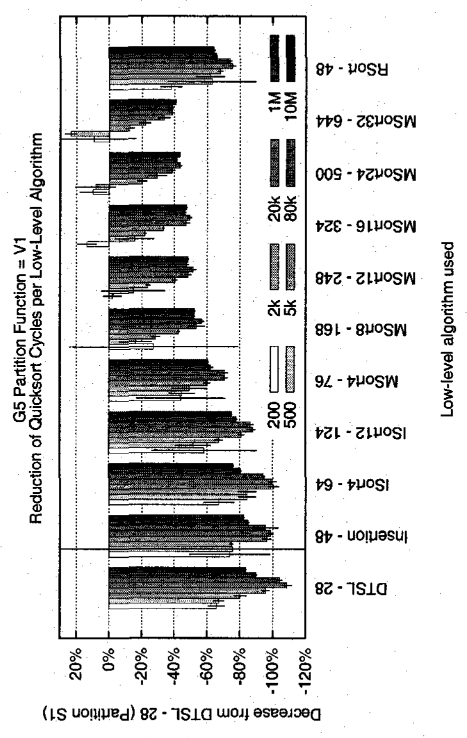
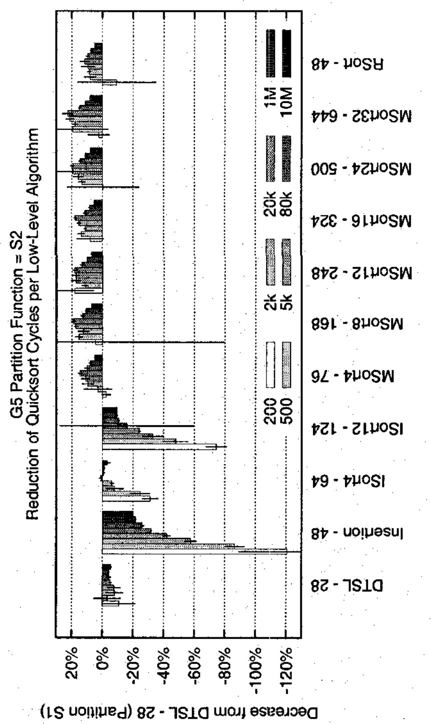
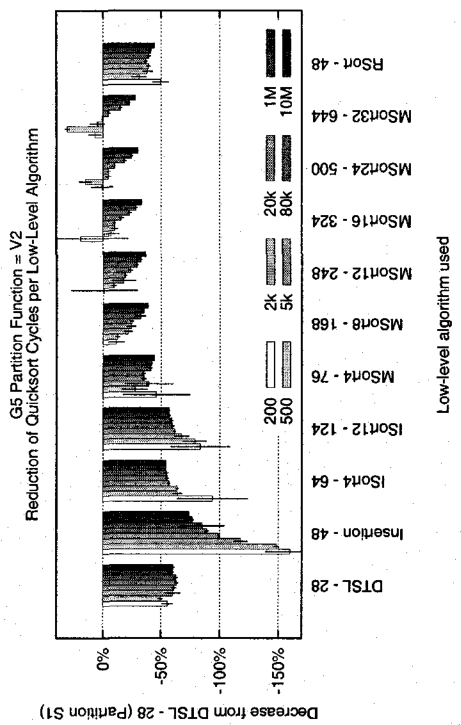
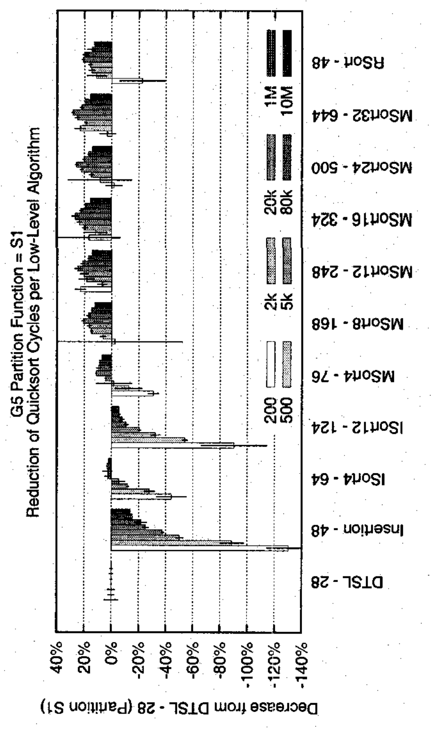


Figure B.9: Reduction in wall-clock time relative to the baseline on a G5. Keys are initially in decreasing order. Error bar is 1 standard deviation.

G5 – Pipe Organ Distribution







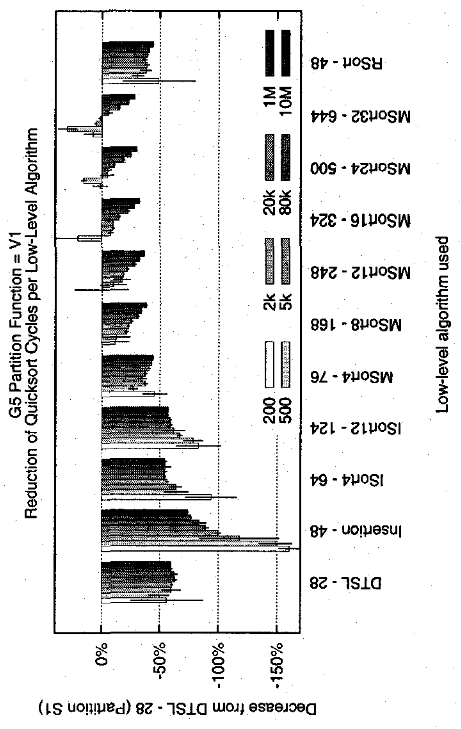
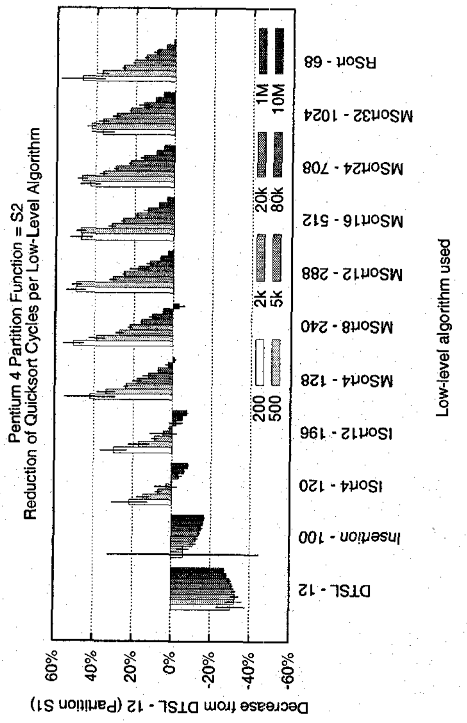
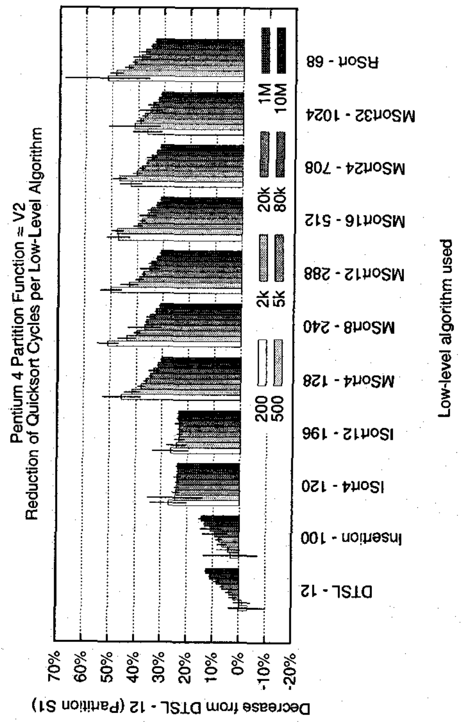
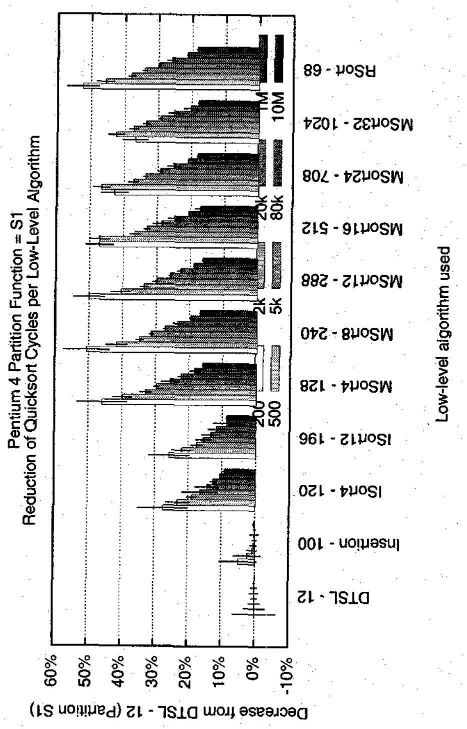


Figure B.10: Reduction in wall-clock time relative to the baseline on a G5. Keys are initially increasing/decreasing with a peak at the middle. Error bar is 1 standard deviation.

Pentium 4 – Large Uniform Distribution







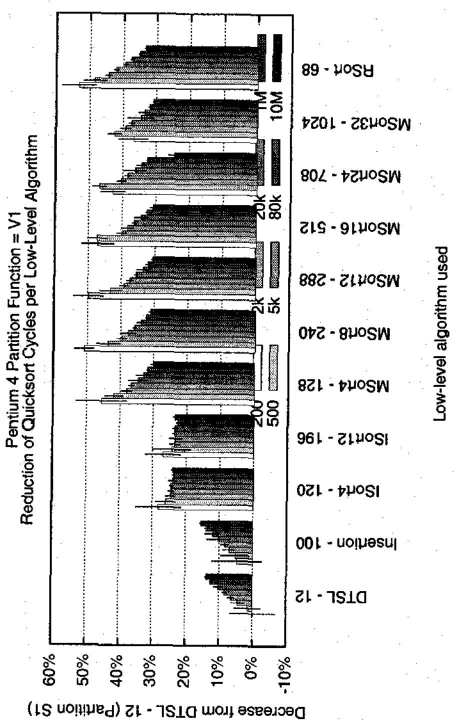


Figure B.11: Reduction in wall-clock time relative to the baseline on a Pentium 4. Keys are drawn from a uniform distribution with a large range. Error bar is 1 standard deviation.

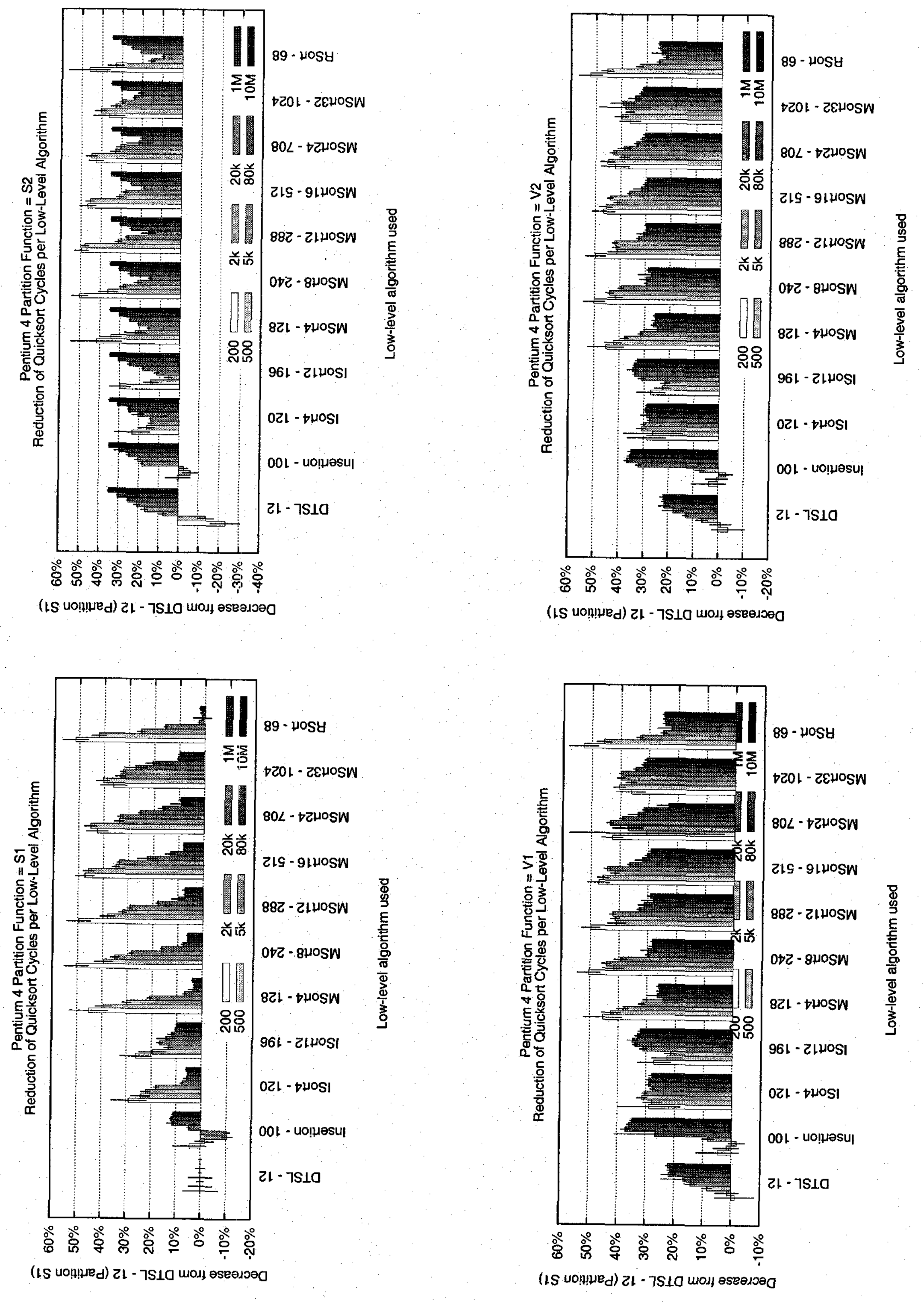


Figure B.12: Reduction in wall-clock time relative to the baseline on a Pentium 4. Keys are drawn from a uniform distribution with a range of 100. Error bar is 1 standard deviation.

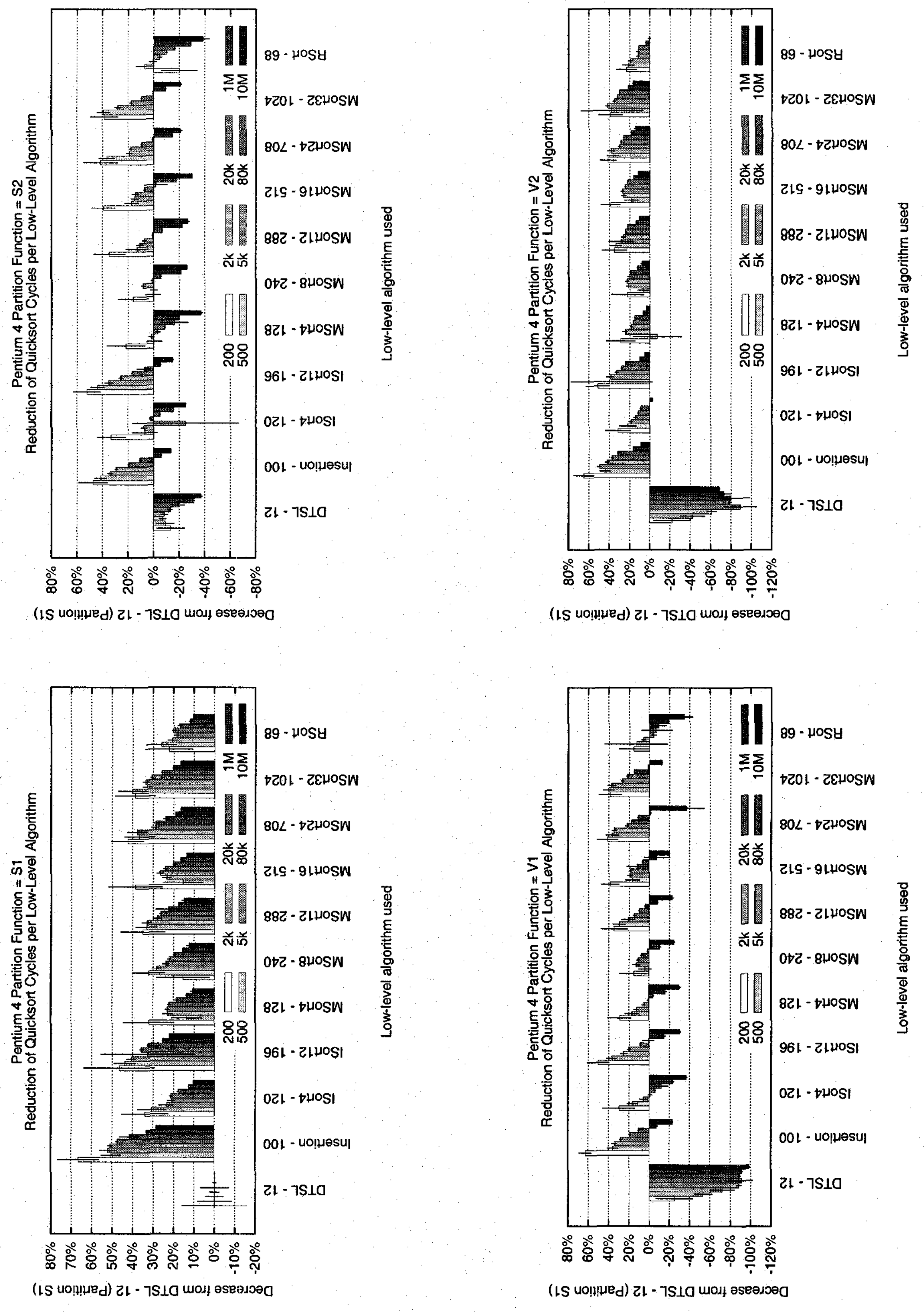


Figure B.13: Reduction in wall-clock time relative to the baseline on a Pentium 4. Keys are initially in increasing order. Error bar is 1 standard deviation.

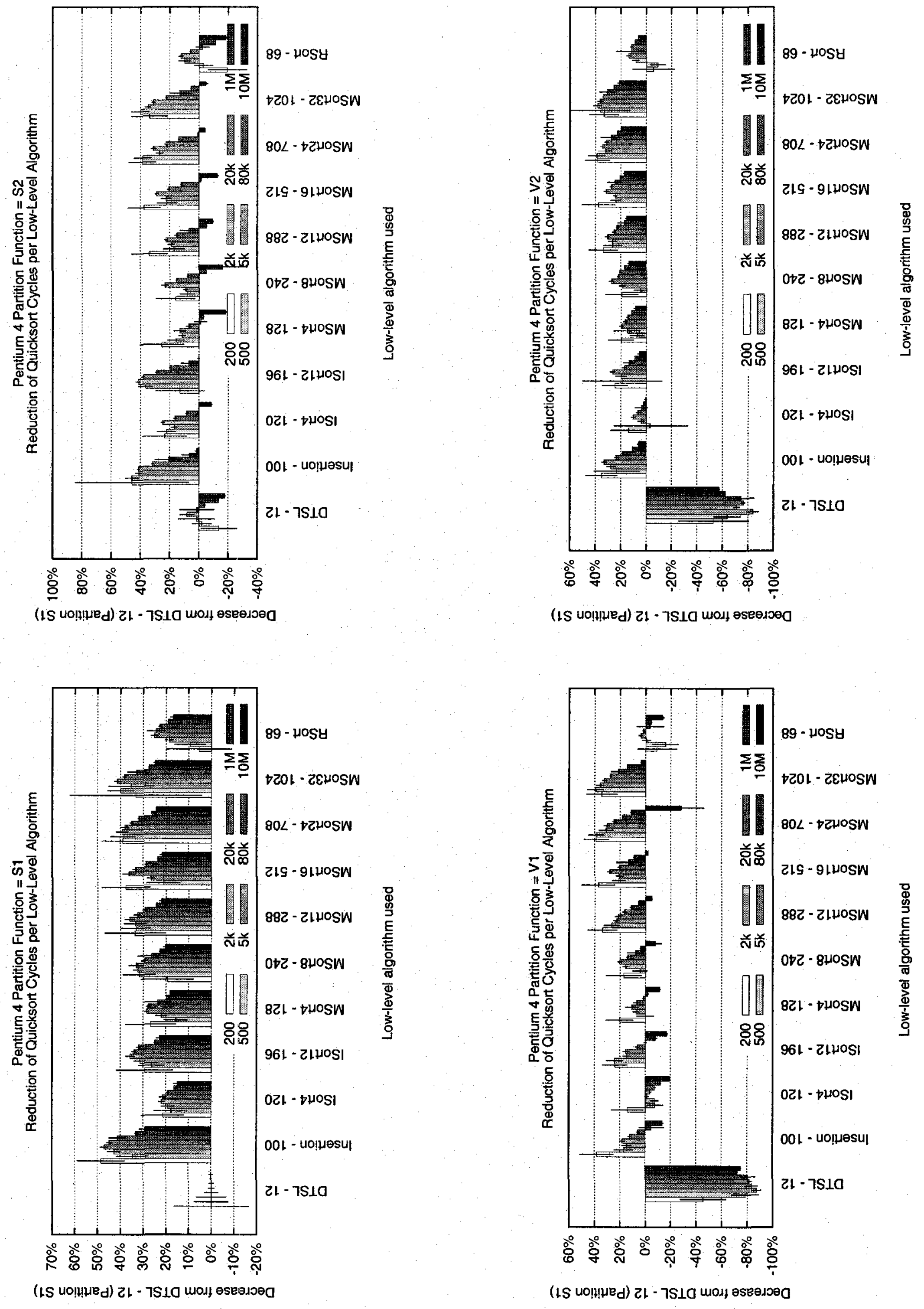
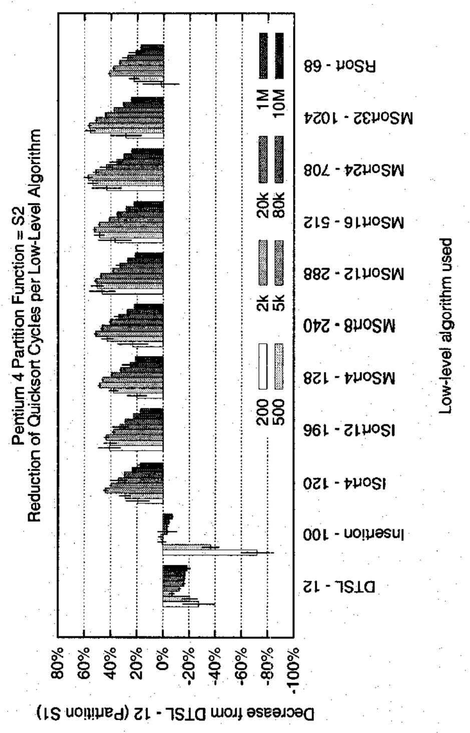
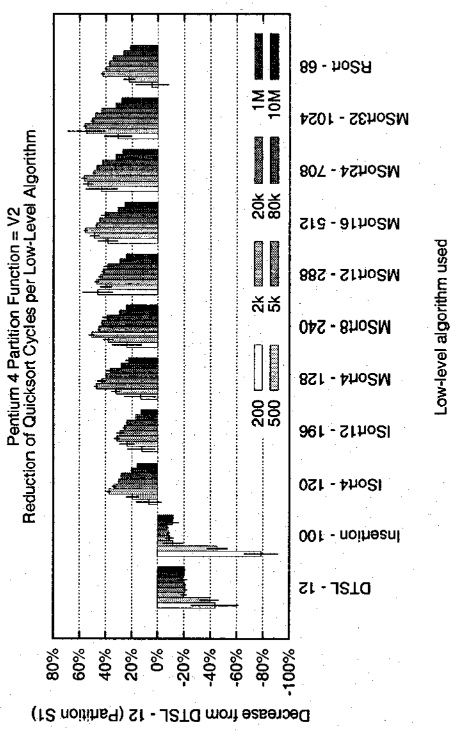
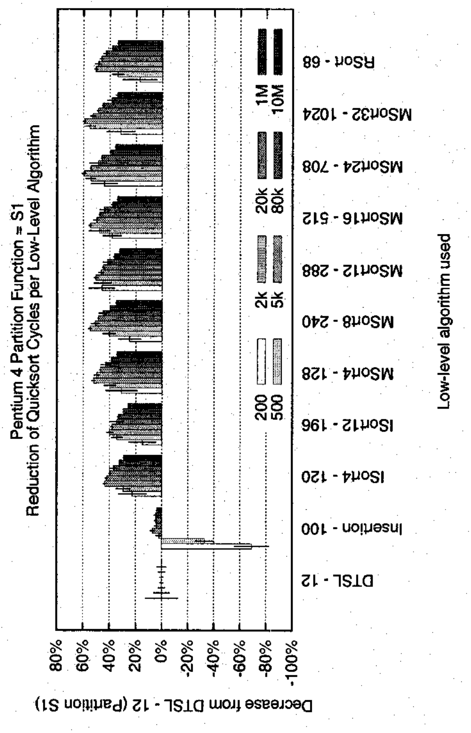


Figure B.14: Reduction in wall-clock time relative to the baseline on a Pentium 4. Keys are initially in decreasing order. Error bar is 1 standard deviation.







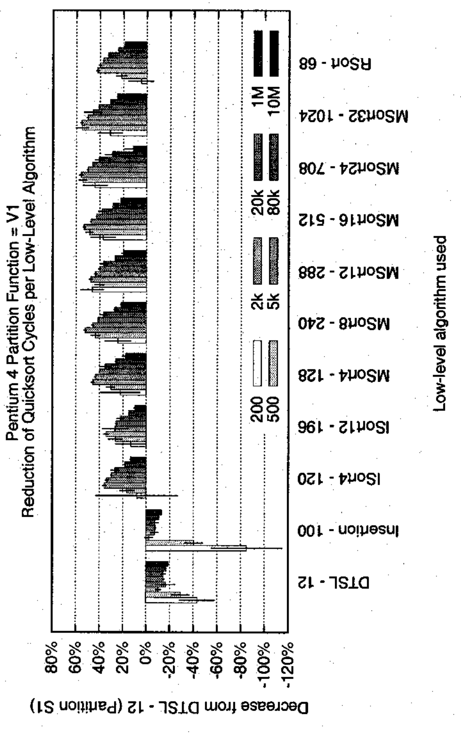
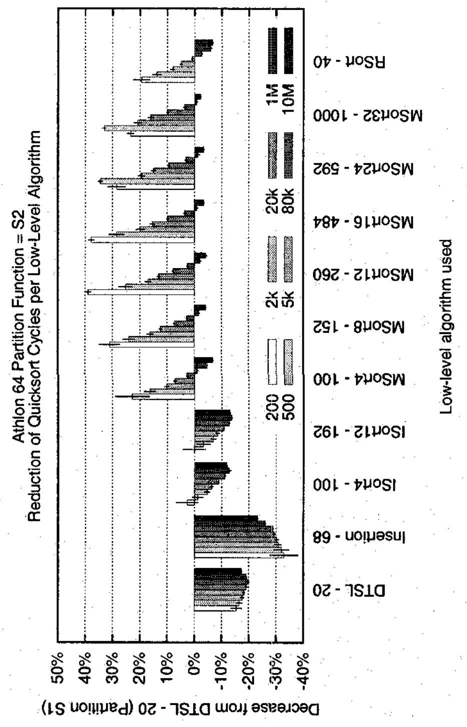
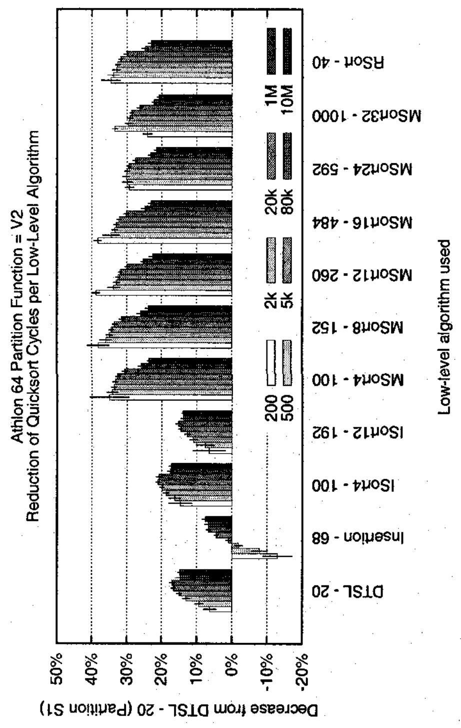
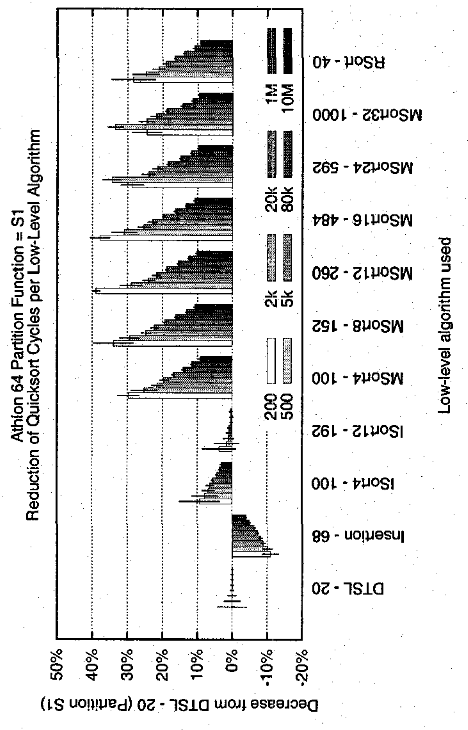


Figure B.15: Reduction in wall-clock time relative to the baseline on a Pentium 4. Keys are initially increasing/decreasing with a peak at the middle. Error bar is 1 standard deviation.

Athlon 64 – Large Uniform Distribution







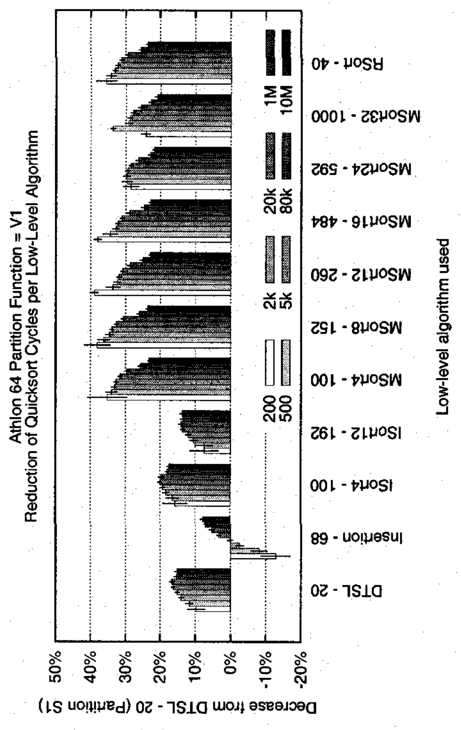
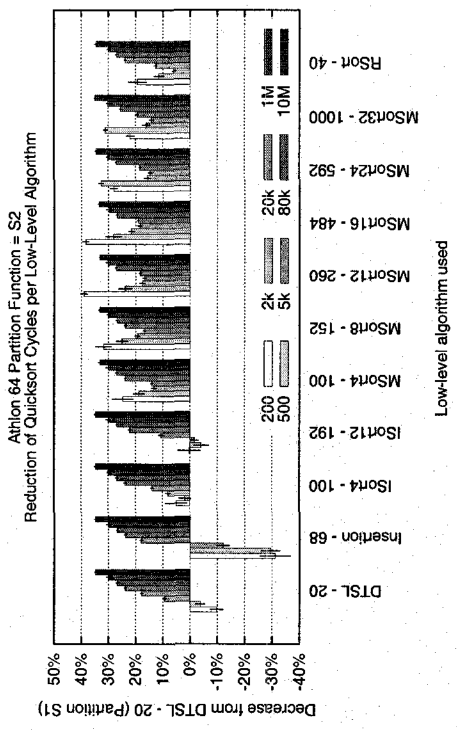
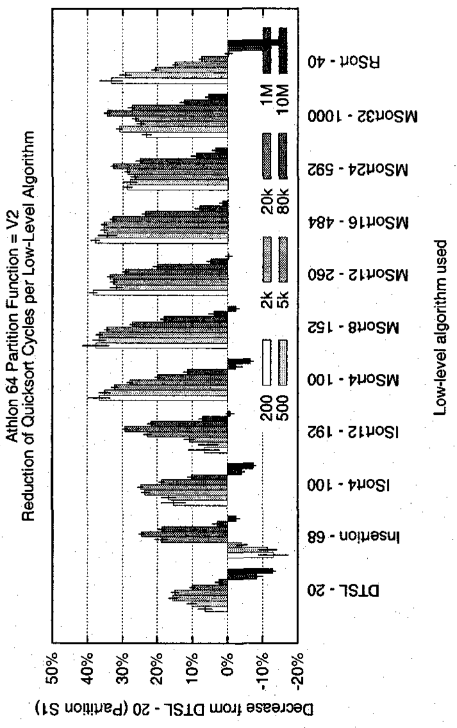
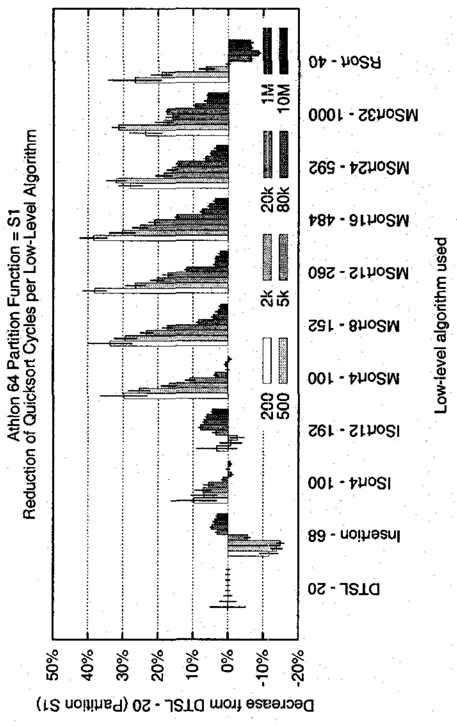


Figure B.16: Reduction in cycle counts relative to the baseline on a Athlon 64. Keys are drawn from a uniform distribution with a large range. Error bar is 1 standard deviation.

Athlon 64 – Small Uniform Distribution







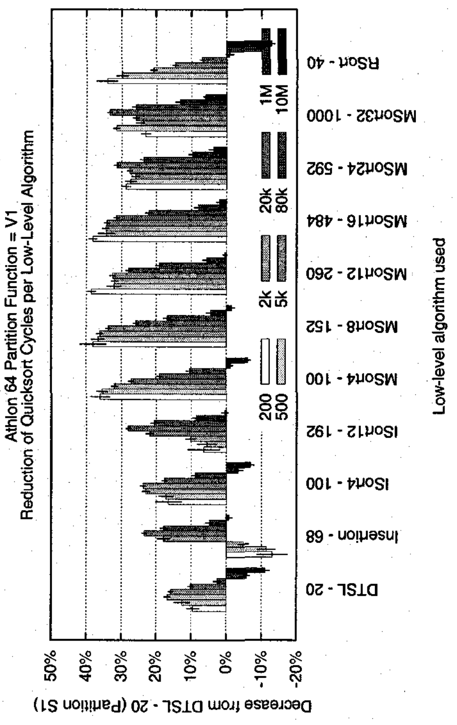


Figure B.17: Reduction in cycle counts relative to the baseline on a Athlon 64. Keys are drawn from a uniform distribution with a range of 100. Error bar is 1 standard deviation.

Athlon 64 – Increasing Distribution

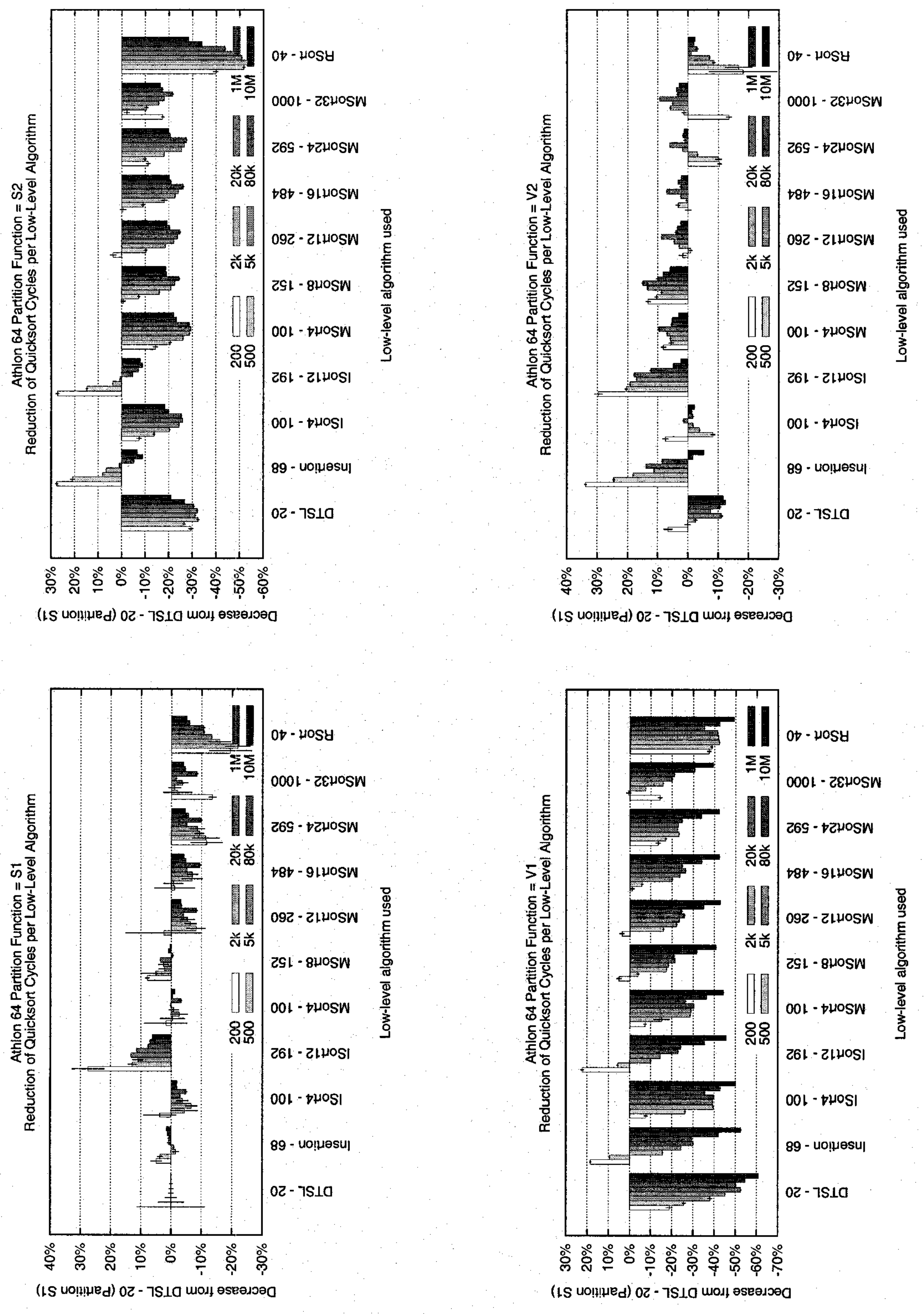


Figure B.18: Reduction in cycle counts relative to the baseline on a Athlon 64. Keys are initially in increasing order. Error bar is 1 standard deviation.

Athlon 64 – Decreasing Distribution

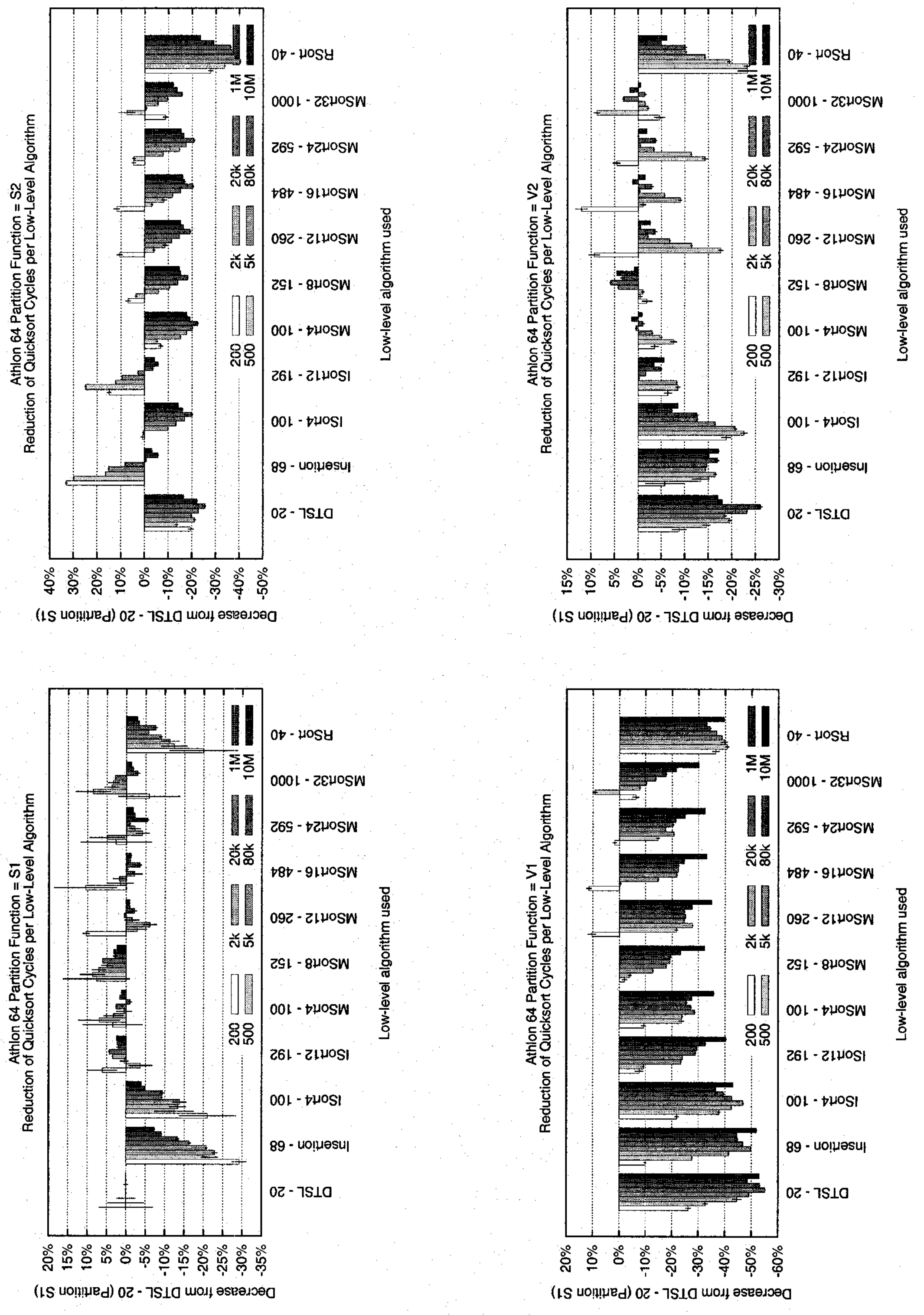
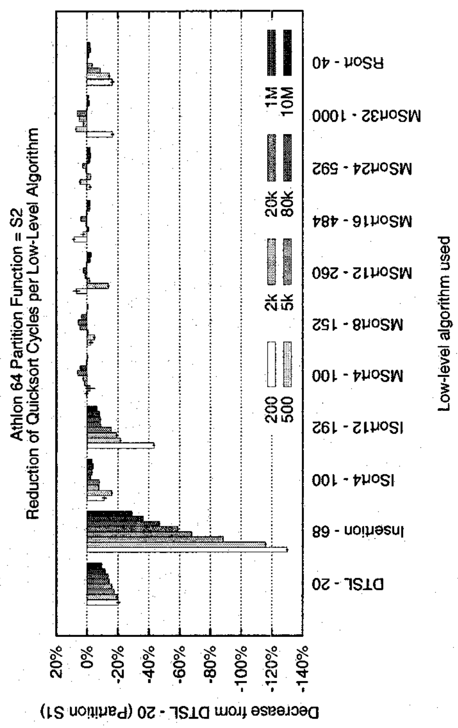
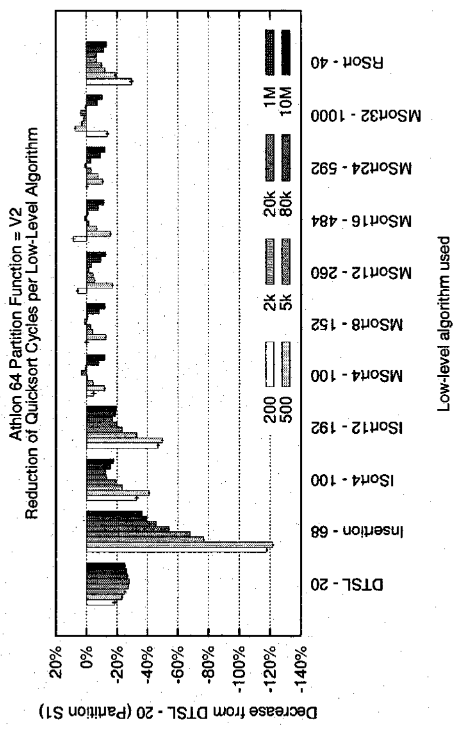
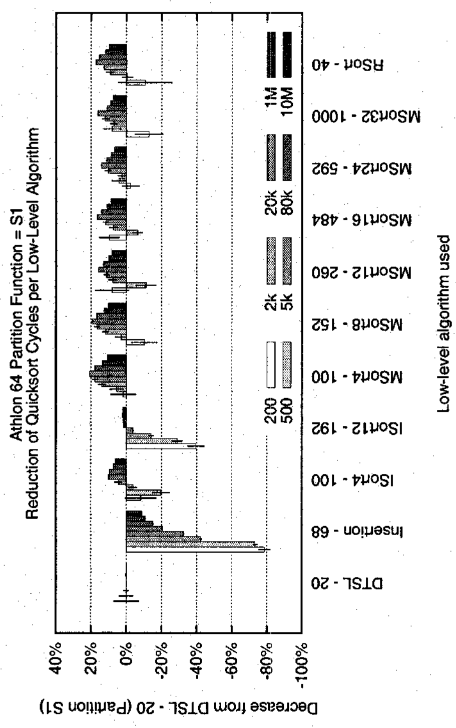


Figure B.19: Reduction in cycle counts relative to the baseline on a Athlon 64. Keys are initially in decreasing order. Error bar is 1 standard deviation.

Athlon 64 – Pipe Organ Distribution







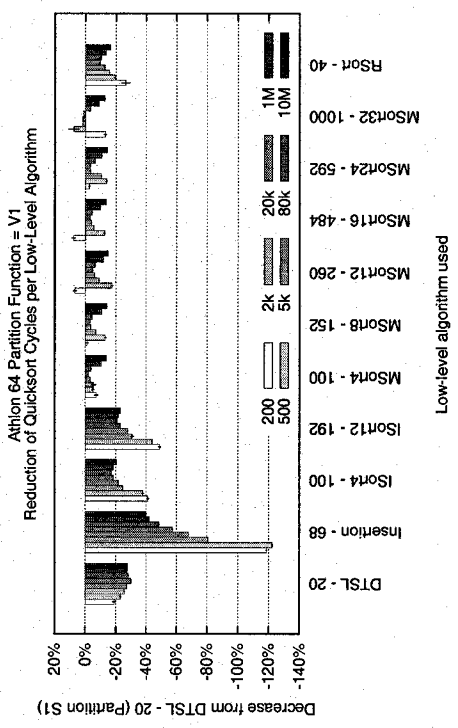


Figure B.20: Reduction in cycle counts relative to the baseline on a Athlon 64. Keys are initially increasing/decreasing with a peak at the middle. Error bar is 1 standard deviation.

Appendix C

Heaps

Heapsort Times for the Core 2 Duo

Array	**Algorithm**																		
Size	2	8	12	16	2F	8F	12F	16F	24F	4V	8V	12V	16V	24V	4FV	8FV	12FV	16FV	24FV
2^4	110	113	118	123	**98**	112	117	112	116	128	119	113	125	124	122	113	106	116	115
2^5	112	123	128	131	**100**	124	136	142	134	144	131	131	130	134	135	124	124	126	128
2^6	121	135	146	146	**106**	139	158	172	160	155	134	142	144	153	145	127	131	139	140
2^7	133	155	159	162	**116**	154	180	198	185	167	147	145	151	161	157	138	135	142	156
2^8	145	172	179	178	**127**	176	203	219	209	186	159	157	152	164	174	150	143	142	159
2^9	159	189	201	201	**140**	197	230	249	228	201	167	170	165	165	190	156	158	156	157
2^{10}	174	209	224	226	**154**	217	260	285	248	222	181	180	180	175	210	170	166	171	168
2^{11}	188	231	244	249	**168**	242	284	320	280	242	199	185	190	192	229	188	171	181	187
2^{12}	204	251	268	271	**183**	266	313	347	315	264	211	205	195	206	251	199	192	187	200
2^{13}	228	276	295	298	210	291	350	380	355	289	232	227	216	218	276	220	215	**209**	212
2^{14}	258	304	323	330	239	323	383	423	386	319	258	243	242	227	307	245	231	235	**222**
2^{15}	291	332	348	358	273	353	413	463	426	348	276	259	259	253	338	264	**249**	253	**249**
2^{16}	324	359	380	386	307	382	451	497	471	381	302	287	272	279	369	288	279	**266**	275
2^{17}	362	392	413	415	348	415	490	536	518	412	332	312	299	298	401	317	303	**292**	296
2^{18}	429	438	453	461	431	476	543	596	575	466	367	344	343	**333**	458	353	337	339	**333**
2^{19}	617	515	544	557	634	556	634	695	667	564	446	420	431	420	566	442	**419**	431	420
2^{20}	919	644	662	660	933	691	758	809	786	714	577	537	531	535	706	571	530	**525**	539
2^{21}	1282	781	786	776	1288	826	887	925	908	874	705	644	643	647	873	697	647	**640**	652
2^{22}	1657	929	912	900	1684	966	1021	1059	1037	1050	841	756	757	**749**	1054	830	754	762	775

Table C.1: Heapsort times for the Core 2 Duo, uniform distribution, normalized by array size.

Heapsort Times for the Pentium 4

Array	**Algorithm**																	
Size	2	8	12	16	2F	8F	12F	16F	24F	4V	8V	12V	16V	24V	4FV	8FV	12FV	16FV	24FV
2^4	**0.07**	0.08	0.09	0.09	0.09	0.08	0.08	0.10	0.09	0.11	0.11	0.09	0.11	0.09	0.10	0.10	0.08	0.09	0.08
2^5	**0.08**	0.10	0.10	0.11	0.09	0.09	0.09	0.10	0.11	0.11	0.12	0.11	0.11	0.10	0.12	0.11	0.10	0.10	0.10
2^6	**0.09**	0.11	0.12	0.12	**0.09**	0.10	0.10	0.11	0.12	0.12	0.12	0.12	0.11	0.13	0.12	0.11	0.12	0.11	0.12
2^7	0.10	0.12	0.13	0.13	**0.09**	0.11	0.12	0.12	0.13	0.12	0.12	0.13	0.12	0.14	0.12	0.12	0.12	0.11	0.13
2^8	0.11	0.14	0.14	0.14	**0.10**	0.13	0.14	0.13	0.15	0.14	0.13	0.13	0.12	0.14	0.13	0.11	0.12	0.11	0.14
2^9	0.12	0.15	0.16	0.16	**0.11**	0.14	0.15	0.15	0.17	0.15	0.12	0.13	0.12	0.15	0.14	**0.11**	0.12	**0.11**	0.14
2^{10}	0.14	0.17	0.18	0.17	0.13	0.16	0.17	0.16	0.18	0.16	0.13	0.13	**0.12**	0.15	0.16	**0.12**	**0.12**	**0.12**	0.14
2^{11}	0.16	0.18	0.20	0.19	0.14	0.17	0.19	0.19	0.20	0.18	0.14	0.13	0.13	0.15	0.17	0.13	**0.12**	**0.12**	0.14
2^{12}	0.17	0.20	0.22	0.22	0.16	0.20	0.21	0.21	0.23	0.20	0.15	0.14	0.13	0.15	0.19	0.14	0.13	**0.12**	0.14
2^{13}	0.19	0.22	0.24	0.23	0.18	0.21	0.23	0.22	0.25	0.21	0.16	0.15	0.14	0.15	0.20	0.15	0.15	**0.13**	0.14
2^{14}	0.21	0.24	0.27	0.25	0.20	0.23	0.26	0.25	0.27	0.23	0.18	0.16	0.15	0.15	0.22	0.17	0.15	**0.14**	**0.14**
2^{15}	0.23	0.27	0.29	0.28	0.22	0.26	0.28	0.27	0.29	0.25	0.19	0.17	0.16	0.17	0.24	0.18	0.16	**0.15**	0.16
2^{16}	0.26	0.29	0.31	0.30	0.25	0.28	0.30	0.29	0.32	0.27	0.20	0.19	0.17	0.18	0.27	0.19	0.18	**0.16**	0.18
2^{17}	0.29	0.32	0.34	0.32	0.28	0.31	0.33	0.31	0.35	0.31	0.23	0.21	0.19	0.20	0.30	0.22	0.20	**0.18**	0.19
2^{18}	0.34	0.35	0.37	0.36	0.33	0.34	0.37	0.35	0.38	0.34	0.25	0.23	0.22	0.22	0.34	0.24	0.22	**0.21**	**0.21**
2^{19}	0.44	0.40	0.43	0.41	0.43	0.40	0.42	0.40	0.44	0.41	0.30	0.29	**0.26**	0.27	0.41	0.30	0.27	**0.26**	**0.26**
2^{20}	0.58	0.48	0.49	0.48	0.58	0.47	0.49	0.47	0.53	0.48	0.37	0.35	**0.32**	0.34	0.48	0.36	0.34	**0.32**	0.33
2^{21}	0.76	0.55	0.57	0.54	0.74	0.55	0.56	0.54	0.61	0.56	0.44	0.42	0.39	0.41	0.57	0.44	0.41	**0.38**	0.40
2^{22}	0.93	0.63	0.64	0.61	0.93	0.63	0.64	0.61	0.70	0.66	0.52	0.49	0.46	0.48	0.66	0.52	0.48	**0.45**	0.46

Table C.2: Heapsort times for the Pentium 4, uniform distribution, normalized by array size.

Heapsort Times for the Athlon 64

Array Size	Algorithm																		
	2	8	12	16	2F	8F	12F	16F	24F	4V	8V	12V	16V	24V	4FV	8FV	12FV	16FV	24FV
2^4	120	137	151	154	**113**	134	148	136	146	143	137	139	158	156	137	137	134	149	149
2^5	124	144	160	169	**115**	145	167	172	168	151	149	153	160	165	148	141	152	158	164
2^6	140	161	175	191	**129**	158	187	206	201	167	148	162	175	192	163	142	157	170	185
2^7	156	170	198	207	**142**	175	211	233	234	186	161	167	184	211	180	156	163	178	200
2^8	174	197	216	227	**157**	198	233	258	262	207	178	180	187	223	203	172	176	183	209
2^9	192	221	232	251	**175**	224	264	287	284	231	193	200	207	229	226	185	198	202	222
2^{10}	209	244	260	300	**191**	248	296	325	311	257	211	214	232	248	253	205	214	227	236
2^{11}	228	266	297	312	**209**	276	323	362	352	285	231	227	248	279	281	227	226	246	270
2^{12}	246	293	309	348	**227**	305	353	392	393	311	248	250	261	304	308	242	251	257	295
2^{13}	265	319	350	384	**245**	329	389	424	423	337	267	270	281	322	334	261	274	281	310
2^{14}	294	350	378	418	**276**	361	427	470	454	370	298	292	315	339	366	295	295	311	327
2^{15}	335	389	413	461	**318**	401	474	526	514	416	339	330	352	386	411	333	329	348	373
2^{16}	397	428	473	518	391	451	534	578	574	467	387	381	376	432	469	391	**367**	373	418
2^{17}	576	515	565	610	590	532	628	675	682	558	469	453	455	519	564	477	**452**	455	510
2^{18}	796	617	674	740	827	640	741	814	805	670	564	**537**	557	622	675	572	544	559	626
2^{19}	1057	731	803	889	1075	757	878	975	941	800	684	**638**	656	739	816	688	652	660	752
2^{20}	1342	867	951	1055	1383	886	1037	1136	1120	950	811	**754**	756	889	968	816	772	764	912
2^{21}	1665	984	1098	1219	1698	1013	1181	1301	1290	1105	930	**854**	870	1039	1116	932	872	877	1065
2^{22}	2004	1127	1258	1403	2020	1153	1345	1496	1469	1273	1075	**972**	1002	1201	1282	1067	993	1013	1231

Table C.3: Heapsort times for the Athlon 64, uniform distribution, normalized by array size.

Heapsort Times for the G5

Array Size	Algorithm																	
	2	8	12	16	2F	8F	12F	16F	24F	4V	8V	12V	16V	24V	4FV	8FV	12FV	16FV	24FV
2^4	2.3	2.3	2.4	2.4	2.4	**2.2**	2.3	2.3	2.3	2.6	2.4	2.4	2.4	2.4	2.7	2.3	**2.2**	2.4	2.4
2^5	2.9	2.8	3.1	3.2	8.8	2.8	3.1	3.1	3.0	3.1	2.8	2.8	2.9	2.9	3.0	**2.6**	**2.6**	2.7	2.8
2^6	3.5	3.3	3.9	5.0	3.3	3.2	3.8	4.1	4.0	3.7	3.0	3.1	3.1	3.4	3.5	**2.8**	3.3	3.0	3.2
2^7	4.1	3.8	4.5	4.9	4.1	3.8	4.5	5.0	4.9	4.2	3.3	3.2	3.2	3.7	4.1	3.1	**3.0**	**3.0**	3.5
2^8	4.9	4.4	5.2	5.6	4.7	4.4	5.3	5.6	5.8	4.8	3.8	3.4	3.2	3.9	4.8	3.6	3.2	**3.0**	3.8
2^9	5.6	5.1	6.1	6.4	5.5	5.0	6.0	6.5	6.5	5.4	4.2	3.9	3.5	4.1	5.4	4.0	3.7	**3.4**	3.9
2^{10}	6.3	5.6	6.9	7.4	6.1	5.7	6.8	7.4	7.3	5.9	4.7	4.4	4.1	4.4	5.8	4.5	4.1	**3.8**	4.2
2^{11}	6.9	6.3	7.4	8.3	6.7	6.3	7.4	8.4	8.2	6.4	5.2	4.7	4.4	4.8	6.3	5.0	4.5	**4.3**	4.6
2^{12}	7.6	6.9	8.1	8.9	7.4	6.8	8.2	9.0	9.2	6.9	5.5	5.2	4.7	5.2	6.8	5.3	5.0	**4.6**	5.0
2^{13}	8.4	7.5	9.0	9.8	8.2	7.5	9.1	9.9	10.0	7.3	6.0	5.8	5.3	5.5	7.3	5.8	5.7	**5.1**	5.4
2^{14}	9.2	8.2	9.8	10.8	9.0	8.2	9.9	11.0	10.7	7.9	6.5	6.2	5.8	5.8	7.9	6.4	6.2	**5.7**	5.7
2^{15}	10.4	9.1	11.0	11.9	10.1	9.2	10.9	12.3	11.9	8.6	7.1	6.8	6.7	6.5	8.7	7.1	6.7	**6.4**	**6.4**
2^{16}	12.2	10.6	12.7	13.6	12.2	10.6	13.0	14.1	14.2	10.1	**7.7**	8.4	7.9	8.2	10.2	8.1	8.3	7.9	8.2
2^{17}	16.9	13.4	16.0	17.0	16.6	13.0	16.1	17.2	17.8	13.0	**10.0**	11.5	11.3	11.8	13.2	10.1	11.4	11.3	11.7
2^{18}	22.2	16.6	18.8	20.4	22.8	16.2	19.4	20.7	22.0	16.2	**12.3**	14.5	14.5	15.2	17.1	12.7	14.4	14.6	15.2
2^{19}	31.0	21.1	23.5	24.9	31.1	20.1	23.9	25.0	26.5	22.2	**15.6**	18.6	18.5	18.8	22.4	15.9	18.7	18.7	18.7
2^{20}	41.8	26.2	28.3	29.9	42.9	25.0	28.9	29.9	32.7	27.8	**19.7**	23.0	22.3	22.9	28.5	20.3	22.8	22.5	22.9
2^{21}	54.8	31.8	33.6	35.0	55.2	30.7	34.2	35.3	38.5	34.8	**24.8**	27.3	26.3	26.0	35.0	25.3	27.0	26.3	26.1
2^{22}	68.7	37.7	38.3	40.0	69.5	36.9	39.4	40.5	43.2	42.4	30.6	30.8	29.1	28.3	42.9	31.0	30.9	29.3	**27.9**

Table C.4: Heapsort times for the G5, uniform distribution, normalized by array size.

Appendix D

Source Code

```
//==================================================================

/** Simpler than a comparator, only keep track of the minimum. */

static inline __attribute__((always_inline))
void myswap(__m128 &a, const __m128 &b, __m128 &A, const __m128 &B)
{
  a = _mm_min_ps(a, b);
  __m128 mask = _mm_cmpneq_ps(a, b);
  __m128 mA = _mm_and_ps    (A, mask);
  __m128 mB = _mm_andnot_ps(mask, B);
  A = _mm_or_ps(mA, mB);
}

//------------------------------------------------------------------

/** Doesn't preserve key values, we only need the index. */

static inline __attribute__((always_inline))
void finalswap(__m128 &a, const __m128 &b, __m128 &A, const __m128 &B)
{
  __m128 mask = _mm_cmplt_ps(a, b);
  __m128 mA = _mm_and_ps(A, mask);
  __m128 mB = _mm_andnot_ps(mask, B);
  A = _mm_or_ps(mA, mB);
}

//------------------------------------------------------------------

/** Return the index associated with the smallest key in the input vector. */

static inline __attribute__((always_inline))
int horz_min(__m128 &a, __m128 &A)
{
  __m128 b = _mm_movehl_ps(b, a); // b[0,1] = a[2,3]
  __m128 B = _mm_movehl_ps(B, A); // B[0,1] = A[2,3]

  myswap(a, b, A, B);

  b = (__m128)_mm_shuffle_epi32((__m128i)a, 0x01); // b[0] = a[1]
  B = (__m128)_mm_shuffle_epi32((__m128i)A, 0x01); // B[0] = A[1]

  finalswap(a, b, A, B);

  union foo {
    unsigned int i;
    float f;
  };

  __attribute__((__aligned__(16))) foo final;
  _mm_store_ss(&final.f, A);
  return final.i;
}

//==================================================================
```

Figure D.1: Support functions for finding the minimum element in an array.

```
//=====================================================================

/** Return the index of the minimum element from an array of 12 key-pointer pairs.
    Input is already typecast as float* from an array of key-pointer structs.
*/

static inline __attribute__((always_inline))
int find_min_12i(const float *key_in)
{
  __attribute__((__aligned__(16))) static const unsigned int index[12] =
    { 0,1,2,3,4,5,6,7,8,9,10,11 };

  __m128 a, b, A, B;

  // first 4 key-pointer elements

  a = _mm_load_ps(key_in + 0 * 8 + 0);
  { __m128 q = _mm_load_ps(key_in + 0 * 8 + 4);
  a = _mm_shuffle_ps(a, q, 0x88); }
  A = _mm_load_ps((float*)index);

  // elements 5-8

  b = _mm_load_ps(key_in + 1 * 8 + 0);
  { __m128 q = _mm_load_ps(key_in + 1 * 8 + 4);
  b = _mm_shuffle_ps(b, q, 0x88); }
  B = _mm_load_ps((float*)index + 1*4);
  myswap(a, b, A, B);

  // elements 9-12

  b = _mm_load_ps(key_in + 2 * 8 + 0);
  { __m128 q = _mm_load_ps(key_in + 2 * 8 + 4);
  b = _mm_shuffle_ps(b, q, 0x88); }
  B = _mm_load_ps((float*)index + 2*4);
  myswap(a, b, A, B);


  return horz_min(a, A);
}

//=====================================================================
```

Figure D.2: Vector code to find the minimum element from an array of 12 key-pointer pairs. Uses functions defined in Fig. D.1.

Sizes of C Object Files for Sorting Networks up to a Given Size

Array Size ≤	**.o File Size (bytes)**			
	x86-64 SSort	**x86-64 RSort**	**G5 SSort**	**G5 RSort**
4	1496	1696	980	1916
8	1752	2312	1324	4440
12	2144	3328	1764	8432
16	2632	4584	2272	13540
20	3312	6392	3016	20844
24	4128	8672	3952	29260
28	5192	11976	5304	39900
32	6512	16112	7012	51496
36	8248	21728	9388	66752
40	10536	28136	12456	82492
44	13424	36336	16448	101884
48	16736	45408	20924	121664
52	20840	56552	26456	145088
56	25488	68496	32580	169228
60	30784	82752	39644	197144
64	36616	97320	47432	225404
68	43664	115160	56452	258952
72	51616	133632	66416	292468
76	60488	155368	77512	330616
80	70040	177336	89636	369348
84	80544	202624	103024	412904
88	92040	228296	117356	456540
92	104440	256856	133028	505444
96	117408	285824	149648	554248
100	131720	318416	167756	608164
104	147000	351704	187096	662496
108	163424	388160	207968	722224
112	180432	425072	229692	782076
116	198808	465624	252888	847104
120	218208	506816	277124	912212
124	238640	551664	302780	982556
128	259800	596024	329640	1053008
⋮	⋮	⋮	⋮	⋮
212	1021888	2098624	1270320	3366340
216	1073384	2195496	1334220	3518644
220	1127320	2297720	1399924	3678568
224	1182528	2398496	1466736	3838416
228	1239816	2506096	1536364	4005928
232	1298616	2612504	1607912	4173488
236	1359104	2724864	1681712	4348572
240	1420752	2835824	1757468	4523340
244	1484632	2952312	1835800	4705756
248	1549664	3067392	1916356	4888380
252	1616912	3188464	1999084	5077872
256	1684920	3308280	2083288	5267672

Table D.1: Size of the compiled object files containing all functions up to and including a given maximum number of elements.