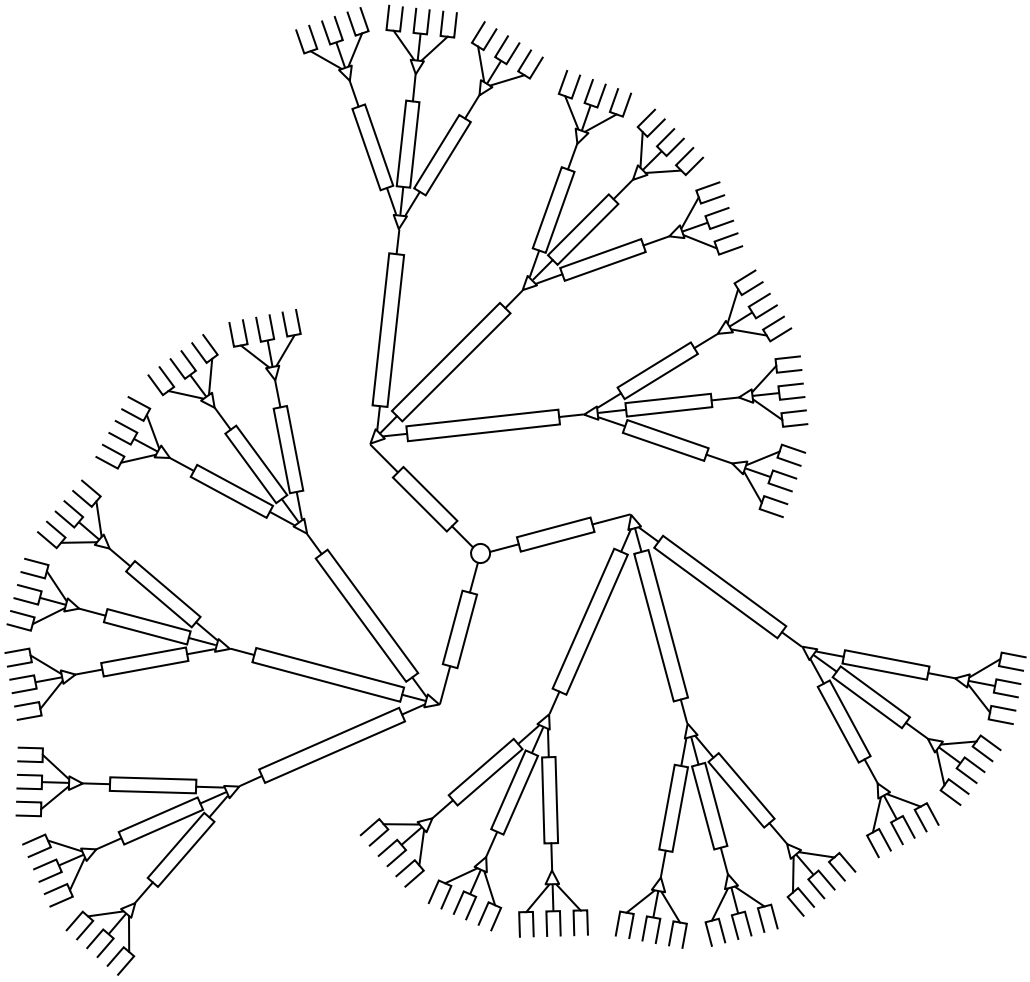
Engineering Cache-Oblivious Sorting Algorithms



Master's Thesis

by

Kristoffer Vinther
June 2003

Abstract

Modern computers are far more sophisticated than simple sequential programs can lead one to believe; instructions are not executed sequentially and in constant time. In particular, the memory of a modern computer is structured in a hierarchy of increasingly slower, cheaper, and larger storage. Accessing words in the lower, faster levels of this hierarchy can be done virtually immediately, but accessing the upper levels may cause delays of millions of processor cycles.

Consequently, recent developments in algorithm design have had a focus on developing algorithms that sought to minimize accesses to the higher levels of the hierarchy. Much experimental work has been done showing that using these algorithms can lead to higher performing algorithms. However, these algorithms are designed and implemented with a very specific level in mind, making it infeasible to adapt them to multiple levels or use them efficiently on different architectures.

To alleviate this, the notion of cache-oblivious algorithms was developed. The goal of a cache-oblivious algorithm is to be optimal in the use of the memory hierarchy, but without using specific knowledge of its structure. This automatically makes the algorithm efficient on all levels of the hierarchy and on all implementations of such hierarchies. The experimental work done with these types of algorithms remain sparse, however.

In this thesis, we present a thorough theoretical and experimental analysis of known optimal cache-oblivious sorting algorithms. We develop our own simpler variants and present the first optimal sub-linear working space cache-oblivious sorting algorithm. These algorithms are implemented and highly optimized to yield high performing sorting algorithms. We then do a thorough performance investigation, comparing our algorithms with popular alternatives.

This thesis is among the first to provide evidence that using cache-oblivious algorithm designs can indeed yield superior performance. Indeed, our algorithms are able to outperform popular sorting algorithms using cache-oblivious sorting algorithms.

We conclude that cache-oblivious techniques can be applied to yield significant performance gains.

Contents

Chapter 1

Introduction

Sorting algorithms are perhaps the most applied, well studied, and optimized of algorithms in computer science; however, there is a notable lack of experimental results when it comes to algorithms designed for the cache-oblivious model. This thesis is a study of the feasibility of algorithms designed for the cache-oblivious model in the context of sorting.

This chapter provides an overview of the relation between theoretical and experimental algorithm analysis, and gives insight in, to what degree popular theoretical tools can give accurate results, why and why not, and establishes newly developed tools, that aim to mend the shortcomings of the more popular ones.

1.1 Algorithm Analysis

An important goal of algorithm design is *efficiency*. When an algorithm is said to be efficient it often refers to the algorithm being fast in the sense that it requires no more computational work to complete than necessary. There are at least three established ways to argue that an algorithm is efficient [Joh01]: through experimental analysis, worst-case analysis, or average-case analysis. The first being a practical approach and the latter two being purely theoretical. Each has its own merits and shortcomings; experimental analysis mimics real-world applications of the algorithm but there are often many factors not relating to the algorithm that pollute the result. While worst-case analysis provides very useful and insightful guaranties of the maximum amount of time the algorithm takes to execute, it may not resemble typical execution times. Average-case analysis attempts to capture typical execution times but provide no guaranties other than the specific case of uniform input.

Experimental results are often gathered from benchmarks that consist of making a computer work as hard as it can on hopefully representative problems, and simply measure the time it took to solve them on a physical clock. The argument here is that if the problems used in the benchmark are the same as or in some sense close to the ones used in real life, the times measured will be close to the time it takes to run the algorithm. If the execution time depends on one or more parameters, such as the input size, the dependency may be extrapolated through usual statistical methods. However, the result may be highly dependent on the hardware on which the benchmark is run.

On the other hand, theoretical analysis seeks to extract properties of the algorithm that are independent of what hardware should be chosen to run it on. To that end theoreticians use *computational models* to approximate the work done by the algorithm,

often focusing on a single type of operation performed, e.g. floating-point operations or comparisons. The result of the analysis is then correct for all computers that work like the model. Furthermore, computational models allow for proof of lower bounds of the time it will take to run *any* algorithm that solves a given problem. Along with a worst-case analysis, this can lead to a proof that a particular algorithm is, short of a constant factor, the *best possible*.

Designing a good model is a non-trivial balancing act; aside from having to resemble the complex inner workings of a typical computer accurately, it also needs to be sufficiently simple to make the analysis feasible. The model must be sufficiently accurate; results obtained from a model that has nothing to do with an actual real-life computer, have no practical relevance.

1.1.1 The Random-Access Machine Model

The most popular model for describing and analyzing algorithms has been the Random-Access Machine (RAM) model [Sav98]. Its chief virtues are that it is very simple and that it indeed seems to behave like typical computers. It states that elements can be stored and retrieved from anywhere in the memory of the computer in unit time and all operations on machine words take unit time regardless of the size of the word. This allows us to analyze the runtime by simply counting the operations performed by the algorithm. Combined with asymptotical analysis, this can lead to very precise statements that are sufficient and relevant for most practical purposes.

Let us, for example, consider a simple algorithm, namely one that computes the product of two matrices A and B. For simplicity, let A be a $1\times n$ matrix and B be an $n\times 500$ matrix. A function written in C that computes the product could look like this:

Algorithm 1-1. `void mprod(int n, const float *A, const float *B, float *u)`

```
{
    for( int i=0; i!=500; ++i )
    {
        u[i] = 0.0f;
        for( int j=0; j!=n; ++j )
            u[i] += A[j]*B[j*500+i];
    }
}
```

The actual runtime of this function depends on how long it takes the computer to do index calculations, comparisons, integer increments, floating-point additions and multiplications, and many other factors. Some hardware is capable of performing several floating-point additions and multiplication at once, which we would have to consider also.

In an asymptotical analysis in the RAM model, however, we can simply say that for sufficiently large n, the runtime is proportional to n. This is because all of the above-mentioned operations take unit time, and that for large enough n, the time to set up the loops are negligible. Now, if the model is accurate, the result should match that of any experimental analysis. A benchmark that measures the average execution time of the function over 30 invocations with n ranging from 50 to 540, run on a 175MHz MIPS R10000 processor gave the result shown in Figure 1-1.

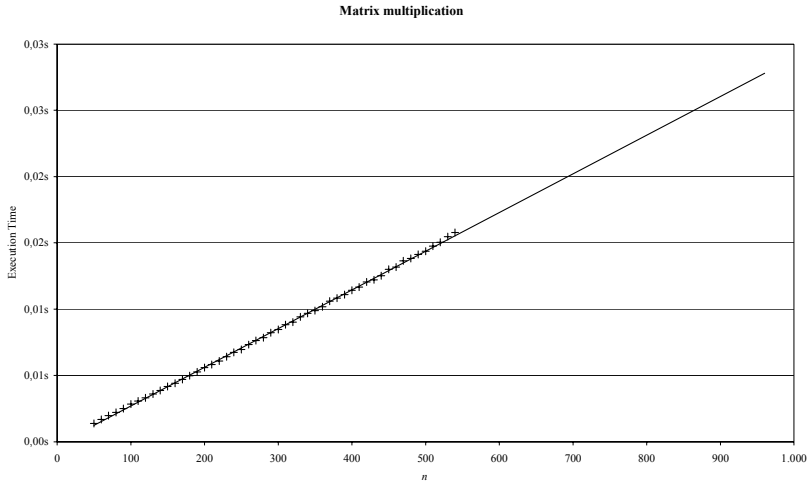


Figure 1-1: Average execution time of matrix multiplication.

We can see that the analysis carried out in the RAM model seem to be in correspondence with what is observed in practice on a MIPS R10000. If the model is correct, the execution time is always linear and should thus continue along the trend line. The power of a good theoretical understanding of an algorithm also lies in an ability to predict the behavior of the algorithm under different circumstances, e.g. change of input parameters. Figure 1-2 shows what happens when even larger matrices are multiplied, the trend line being the same as in Figure 1-1.

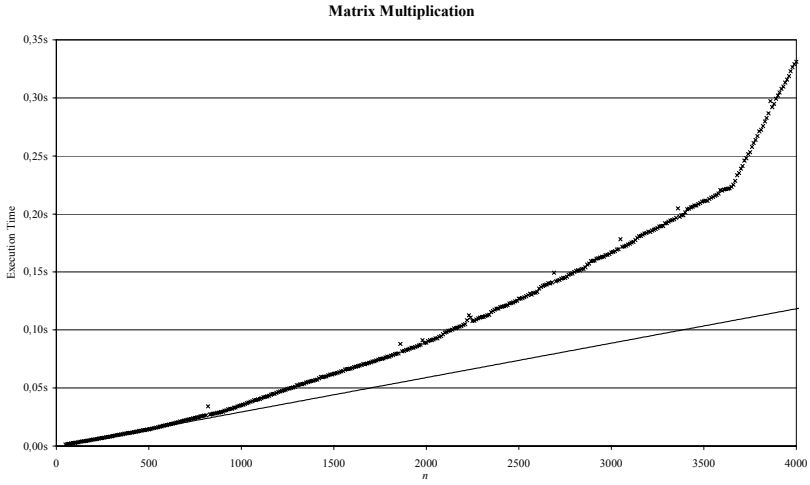


Figure 1-2: Average execution time of matrix multiplication for larger n.

It is clear that the analysis does not correctly describe and predict the real world execution time. Either the analysis is incorrect the model does not reflect reality adequately. It turns out the latter is the case.

1.1.2 Storage Issues

Consider the statement that an analysis is correct for all sufficiently large n. If the algorithm deals with at least n elements, these elements need to be stored somewhere. However, a very real practical issue applies here, namely that storage space in computers is limited. A statement, which applies for all sufficiently large problem sizes, cannot be true in practice; the problem size cannot exceed the capacity of the computer.

Savage formalizes the memory of the RAM as having $m = 2^{\mu}$ storage locations each containing a b-bit word, with μ and b integers [Sav98], so we may revise the statement to say, the RAM model is accurate for all sufficiently large n, but no larger than 2^{μ}. As it became necessary to be able to work with larger sets of data, engineers developed means for supporting this by using layers of storage. We will describe these ideas in the next chapter. Suffice it to say, that it is the effect of these layers, we see influencing the runtime in Figure 1-2. We may then say that $\mu = 14$ for Algorithm 1-1 and revise the conclusion of the analysis accordingly, however that severely limits the power of the analysis. Only if μ was large enough to cover all practical applications of our algorithm could we get a relevant result from the analysis.

An algorithm that can store the problem in one layer of storage can be accurately described by the RAM model. However, when an algorithm begins to make use of the next layer of storage, the execution times start to deviate from what the model predicts. One may argue that memory access still takes unit time for some suitable unit, but it only does so, on exactly one level. If we were to make any guaranties about the execution time, we would have to assume that *all* memory accesses might take the unit of time it takes on the slowest level. Since this might involve operating a mechanical arm to get to magnets on a rotating disk, the unit might be several millions times greater than that of any other operation. The RAM model may be hap hazardously forced to describe algorithms in this way, but doing so will never bring us any insights into problems of dealing with a memory hierarchy so that we may alleviate them.

1.2 Sorting in the Memory Hierarchy

In light of the fact that the analysis of some algorithms in the RAM model may not reflect their real-life performance it is not clear whether algorithms designed to be efficient in the RAM model, are indeed so. Some algorithms may not exhibit the behavior illustrated in Figure 1-2 or may do so, but to a lesser degree. In particular, we are in this thesis interested in the behavior of sorting algorithms, since they are the foundation upon which many other algorithms are built.

As will be described in Chapter 3, computational models have been developed specifically to take into account, the structure of storage of modern computers. In addition, sorting algorithms have been developed that are proven optimal in the sense of these new models. The External Memory model [AV88], formalized the effects of secondary storage, in a way that allows the algorithms to use the parameters that describes the storage. These algorithms are known as *cache-aware* algorithms. Conversely, *cache-oblivious* algorithms are analyzed in essentially an external memory model, but are unaware of the storage parameters. They are optimal in the External Memory model, while they are designed for the Random Access Memory model.

Being cache-oblivious first seems to be a disadvantage; a cache-oblivious algorithm can clearly not be more efficient than the optimum cache-aware algorithm. However, cache-aware algorithms are in practice often designed with two specific levels of the memory hierarchy in mind, making them suboptimal on any other level. They often need to be implemented with specific knowledge of the parameters describing these two levels – knowledge that is not in general available – leading to implementations that are only optimal on those two levels. This is in a way similar to Algorithm 1-1 being limited to multiplying $1 \times n$ with $n \times 500$ matrices and not with general $n \times m$ matrices.

Cache-oblivious algorithms do not suffer from these shortcomings; they are optimal on a level of the hierarchy, regardless of the parameters describing it, which automatically makes them optimal on any level of the hierarchy. It gives them the ability to adapt to changes in the environment, be it due to a memory upgrade, other processes needing storage, or an upgrade to an operating system with a more aggressive memory management policy. In addition, users of the algorithm sees just another sorting algorithm; no need to tell it e.g. how much memory there will be in the computer it is running on. Optimal RAM sorting algorithms enjoy these features too; however, they are not in general optimal in the use of the memory hierarchy. On the other hand, optimal cache-oblivious algorithms are in general more complex than their optimal RAM counterparts are, and the higher memory performance may not make up for the increased instruction count. With sorting in particular, very popular algorithms exist with extremely low instruction count, namely quicksort, mergesort, and heapsort. They are all cache-oblivious algorithms, albeit not optimal in the memory hierarchy. Indeed, it turns out, that at least quicksort and mergesort come very close to also being optimal in the use of memory.

So, is it feasible to employ cache-oblivious algorithms for sorting compared to algorithms that have detailed knowledge of the memory system, and compared to classic sorting algorithms designed for RAMs to have low instruction count, and optimized and tuned over several decades?

1.3 Previous Work

It is widely accepted that quicksort [Hoa61] is the fastest comparison based sorting algorithm on typical datasets. Sedgewick did a thorough analysis and suggested several improvements to lower the instruction count, including using a final insertion sort pass, instead of using quicksort all the way to the bottom of the recursion [Sed78]. More recently, Musser presented a worst-case optimal variant of quicksort, dubbed introsort, using heapsort as a fallback, in case quicksort is going quadratic [Mus97].

LaMarca and Ladner further improves on Sedgewicks quicksort to get better cache performance by doing the insertion sort at the bottom of the recursion, while elements are in cache [LL99]. In the same paper, they present memory optimizations for most all popular RAM sorting algorithms, making them cache-aware, however. Most implementations of the sorting algorithm `std::sort` in The Standard Template Library (STL), a part of the C++ programming language standard, now incorporate optimizations from [Sed78], [Mus97], and [LL99]. [XZK00] improves on their result by taking into account direct mapped caches and translation look-aside buffers. [ACV+00] presents a cache-aware R-merge sort algorithm utilizing registers, that is

superior to the mergesorts of [LL99], while [RR00] presents cache-aware improvements of flashsort [Neu98]. All four articles back their claims with experimental results and are able to show significant improvements. [XZK00] and [ACV+00] does an experimental comparison of their algorithms with those of [LL99], however [LL99] sets out to only demonstrate improvements and thus compares algorithms with reference implementations, not highly tuned ones typically found in standard libraries.

TPIE [TPI02] and LEDA [NM95] with the LEDA-SM extension [CM99] are frameworks for developing cache-aware optimal algorithms. Both provide optimal sorting algorithms.

[FPLR99] concludes the introduction of the cache-oblivious model with experimental results showing stable matrix transposition and matrix multiplication using cache-oblivious algorithms. [LFN02] compares cache-aware static search trees with cache-oblivious trees while [BFR02] presents cache-oblivious dynamic search trees and analyzes them experimentally with different layouts.

[OS02] implements and studies cache-oblivious heaps. For that, they also implement a cache-oblivious sorting algorithm; however, using that algorithm made heap operations up to eight times slower, compared to using the optimized stl::sort, hinting at poor performance of cache-oblivious sorting algorithms. Using stl::sort, performance of the heaps was comparable to cache-aware implementations, though. No other practical studies have been done on cache-oblivious sorting algorithms.

1.4 This Thesis

As discussed in section 1.2, the performance benefits of cache-oblivious algorithms are not at all clear. The goal of this thesis is to investigate the feasibility of using cache-oblivious algorithms for sorting. Some very well established algorithms, most notably quicksort [Hoa61] has been fine-tuned, optimized [Sed78], and refined [Mus97] to achieve very high performance. In order to establish conclusively the feasibility of cache-oblivious sorting algorithms, we must expect to match that level of optimization and fine-tuning.

Many of the above-mentioned uncertainties can only be clarified through thorough experimental analysis. In this thesis, we use practical experimentation as an important guideline for achieving optimal performance.

The focus is on general-purpose generic algorithms, because we seek to maximize applicability. No algorithm will exploit the binary pattern of keys, nor will they use parallel processing or parallel disks. In addition, there will be no programmatic assumption on how elements are stored. The main competitor will be the gold standard of the RAM model, quicksort, which is also comparison based, and memory tuned variants of other algorithms.

1.4.1 Main Contributions

We provide a thorough analysis of a new and improved variant of funnelsort, one of two known optimal cache-oblivious sorting algorithms. The new variant has a number of free parameters and design choices, which are all implemented and explored to yield

an optimized implementation, in the process providing important insights in how to achieve maximal performance from cache-oblivious sorting algorithms.

We also exhibit a novel cache-oblivious sorting algorithm that requires only low-order working space, the Low-order Working Space Cache-oblivious Sorting Algorithm, LOWSCOSA. This is achieved through a space recycling mechanism in merger data structures that can also be applied to optimal external memory sorting algorithms such as multiway mergesort, to reduce their working space requirements.

All algorithms are implemented, optimized, and thoroughly studied through experiments. Finally, feasibility is established through comparison with other popular and fast sorting algorithms, answering the important question: can cache-oblivious algorithms be used as a viable alternative to existing sorting algorithms?

1.4.2 Structure

Chapters 2 and 3 consider the realities of modern day computing from two different perspectives. Chapter 2 presents important aspects of modern processors and the memory hierarchy that are relevant in implementing efficient sorting algorithms, from a hardware architectural point of view, while Chapter 3 presents the theoretical setting in which we may understand how to deal with these new aspects. We formally present the theoretical models designed to capture the effects of the memory hierarchy, along with lower bound on sorting and algorithms that meet that bound. Both chapters provide a basis for the rest of the thesis.

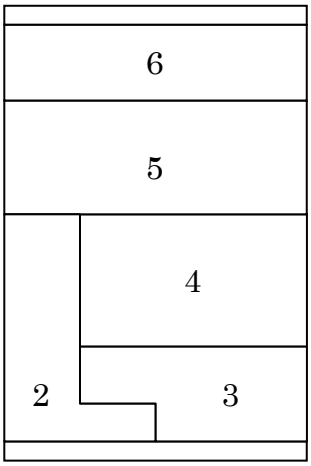


Figure 1-3. Structure of the thesis.

Chapter 4 introduces cache-oblivious sorting algorithms. It exhibits new important simplifications to existing algorithms as well as a thorough analysis of these algorithms. This chapter also exhibits the new optimal sub-linear working space cache-oblivious sorting algorithm.

Chapter 5 is dedicated to the engineering of the algorithms discussed in Chapter 4. Guided by Chapter 2, we develop, optimize, and fine-tune the algorithms to achieve optimal sorting algorithms.

Chapter 6 presents a comparative experimental study of the performance of the developed algorithms and other popular and efficient algorithms. Finally, Chapter 7 provides the conclusion.

Figure 1-3 above illustrates which chapter builds on which; Chapter 3 builds only to a small extend on information provided in Chapter 2, while Chapter 4 continues along the theoretical track of Chapter 3. Chapter 5 builds to a large extend the on the contents of Chapters 2 and 4.

Chapter 2

Modern Processor and Memory Technology

Computation and the storage of data are inseparable concepts. In this chapter, we give a background on how they have evolved and how storage and processors are implemented in computers today. It presents aspects of modern computers that are important for achieving high performance.

We will use knowledge of these aspects when we begin the engineering effort in Chapter 5. We will use it both as guidelines for how to improve performance and to help us understand why performance was or was not improved.

The information provided in this chapter has been gathered from [Tan99], [Tan01], and [HP96] as well as from information available from hardware manufactures on the World Wide Web, such as [HSU+01], [AMD02], and [MPS02]. There will be a lot of detailed information, so the chapter ends in a brief summary of considerations important when implementing high performance sorting algorithms.

2.1 Historical Background

Storage has been the limiting factor on computing, in a much stronger sense than actual raw processing power. Up until the early 1970's computers used magnetic core memory, invented by Jay Forrester in 1949. It was slow, cumbersome, and expensive and thus appeared in very limited quantities, usually no more than 4KB. Algorithms too complex to run in such a limited environment could not be realized and programmers were forced to use simpler and slower algorithms [Tan99, Sect. 6.1]. The solution was to provide a secondary storage. Programmers needing more space for their programs had to split them up manually in so-called overlays and explicitly program loading and storing of these overlays between secondary storage and main memory. Needless to say, much work was involved with the management of overlays.

The situation improved somewhat with the introduction of transistor-based dynamic random-access memory (DRAM, invented at IBM in 1966) by Intel® Corp. and static random-access memory (SRAM) by Fairchild Corp. both in 1970, however high price of memory still made code size an issue. A trend thus arose that useful functionality would be implemented directly in the CPU core; features should rather be implemented in hardware than software. Designs like this, for example, provided single instructions

capable of doing what small loops does in higher level languages, such as copying strings.

Through the 1980's storage of large programs in memory became a non-issue. The complex instruction sets of the 1970's were still used, but they required several physically separate chips to implement (e.g. one for integer operations, one dedicated to floating-point math, and one memory-controller). This complexity prompted a change in design philosophy, toward moving features from hardware to software. The idea was that much of the work done by the more complex instructions in hardware, at the time, could be accomplished by several simpler instructions. The result was the Reduced Instruction Set Computer (RISC). After the RISC term had been established the term Complex Instruction Set Computer (CISC) was used rather pejoratively about computers not following this philosophy.

2.2 Modern Processors

Today's processors can rightfully be called neither RISC nor CISC; at the core, they are in essence all RISC-like. However, whether the processor is based on RISC design or not, have implications we may have to consider to this day. Though the RISC approach is architecturally compelling, the most popular architecture used today is the IA32. The key to its popularity is that it is backwards compatible with all its predecessors, commonly named x86, the first of which was the 8086. The 8086, in turn, was assembly-level compatible with its predecessor, the 8080, which was compatible with Intel's first "computer-on-chip", the 4004. The 4004, 8080, and the 8086 was all designed before the RISC philosophy became prevalent, so the successors of the 8086 are all, what we would call CISC. Today, the x86 processors are the only widely employed CISC processors so they have become synonymous in some sense with the CISC design philosophy. We will use the term RISC to denote processors with roots in the RISC design philosophy, such as the MIPS R4000-R16000, the IBM/Motorola PowerPC, the Digital/HP/Compaq Alpha, and the Sun Ultra Sparc and use the term CISC about processors implementing the IA32.

Common for all processors, RISC or CISC, since the 4004 are the designs with registers, arithmetic/logic unit (ALU) and clocks; each clock tick, operands stored in registers are passed through the ALU, producing the result of some operation performed on them that is then stored in a target register. The particular operation to be performed is decided by the instructions that make up the executing program.

2.2.1 Instruction Pipelining

When looking for ways to enhance performance of a processor, an important observation to make is that each instruction undergoes several steps during its execution. In a typical lifecycle of an instruction it is first fetched from the part of memory the code is stored, then decoded to determine which registers keep the needed operands or if operands are in memory. The instruction is then executed, and finally memory operations are completed and write-back of the result of the operation is completed [HP96, Chap. 3]. These tasks would typically be done in physically and functionally separate parts of the processor, inviting the idea that the work on the next instruction could proceed as soon it has finished the first stage. In computer

architecture, this is known as pipelining. The first pipelined processors were simple ones with only a few stages. This remained the norm until RISC designers started to extend the concept. Figure 2-1 shows the stages of a typical RISC pipeline.

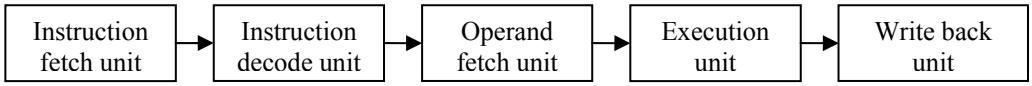


Figure 2-1. A typical RISC five-stage pipeline. While fetching instruction n, this pipeline is decoding instruction n-1, fetching operands for instruction n-2, and so on, all in the same clock cycle.

To make pipelining work efficiently, each stage of the instruction execution must complete in the same time; no instruction can proceed down the pipeline sooner than one can finish an individual stage. While this was one of the primary design goals of the RISC, it cannot be done in case of many of the complex instructions in CISCs. Pipelining can still improve performance, though. The idea is to implement a CISC using a micro-program, in essence emulating the complex instruction set, using a RISC core. This is accomplished by translating the complex instructions into simpler *micro operations* (μ-ops) that each take an equal amount of time to execute. This translation is simply done in additional stages along the pipeline, immediately after the instruction is fetched. Each instruction may result in several μ-ops, so each CISC instruction may take several cycles to complete. Furthermore, while RISC has load-from-/store-to-register instructions and only allows operations to be done on operands in register, CISC has complex addressing modes that allow instructions to operate, not only on operands in registers, but also on operands in memory. To accommodate for this additional address generation stages are inserted in the pipeline. In all, CISC tend to have longer, more complex pipelines than RISC, but the execution proceeds in the same way.

The reason performance is increased using a pipeline approach is that less work is done at each stage, so it can be completed faster. An n-stage pipeline has n instructions working in parallel. This is known as *instruction-level parallelism*. When an instruction proceeds from one stage to the next, it is temporarily stored in a *pipeline latch*, so it does not interfere with the previous instruction. Other than the increased delay incurred by such a latch, there is no reason each stage of the pipeline cannot be split into several simpler, shorter stages, thus making the completion of each stage faster.

Pipeline Hazards

The reason decreasing the complexity of each stage will not improve performance indefinitely, is the increase in the number of stages. Increasing the number of stages, increases the risk of *pipeline hazards*. These are situations, where the next instruction cannot in general proceed, until the previous instruction has either completed, or reached a certain point in the pipeline. There are three kinds of hazards [HP96, Sect. 3.3]:

- *Structural hazards* occur when hardware, such as the memory subsystem, does not support the execution of certain two instructions in parallel.
- *Data hazards* arise when a needed operand is not ready, because either the current instruction is not finished computing it, or it has not arrived from storage.

- *Control hazards* occur when it is simply not known which instruction should come next.

There are several approaches to overcome these problems, the simplest one perhaps being just to halt the pipeline until the hazard has passed. This approach, while expensive in that parts of the processor are simply sitting idle, works relatively well in case of a structural or data hazard. The reason is, these hazards pass in very few cycles. Consider a structural hazard occurring because an instruction is being fetched from memory at the same time another instruction is writing to memory. The memory may not accept that, but stalling the pipeline for one cycle, while one of the instructions proceeds, will resolve the hazard. Likewise will the missing result of one instruction be ready in a few clock cycles, if it is generated by an instruction ahead of it. That instruction must after all be close to completing the execution stage, if the next instruction is ready to execute. Conversely, if the processor is waiting for data from memory, nothing can be done about it in the pipeline.

Control hazards, however, can be much more severe. They typically occur due to branches in the code. Branches are generally indirect, conditional, or unconditional. Unconditional branches simply say that the next instruction is not the one following this one, but another predetermined one. These branches do not pose a problem, since they can be identified quickly and the fetch stage notified accordingly. Indirect branches typically occur when invoking virtual functions (or equivalently calling a function through a function pointer) or when returning from a function. The latter is usually sped up by keeping a dedicated buffer of call-site addresses so when returning from a function, the processor can quickly look up where is to continue. Both types of indirect branches are relatively infrequent so they do not pose a threat to the overall performance, even if the pipeline is stalled for many cycles. Conditional branches are not rare, however.

Algorithm 2-1. `void merge(int n, const float *A, const float *B, float *u)`

```
{
    for( int i=0; i!=n; ++i )
        if( *A < *B )
            *u = *A, ++A, ++u;
        else
            *u = *B, ++B, ++u;
}
```

Consider the simple loop in Algorithm 2-1. After each iteration, a test is done to see if this was the n'th iteration and if not, branch back, and loop again. The test is one instruction and the branch is the one following it, but what should follow the branch instruction? That depends on the result of the test, but the result is not available until the test has gone all the way through the pipeline. Thus, conditional branches may block the fetching of instructions. If we were to make the pipeline longer, it will simply be stalled for a correspondingly longer time on an unconditional branch.

Branch Prediction

Realizing, that simply stalling the pipeline until the result of the test has been determined will make all stages sit idle, we might as well go ahead and proceed along

one of the branches. If it turns out to be the wrong branch, the work done in the pipeline after the branch was futile, and must be discarded (through what is called a *pipeline flush*), but we have wasted no more time than if we had simply stalled the pipeline. On the other hand, if it was the right branch we have wasted no time at all! The next question is what branch to follow? Consider again the loop branch in Algorithm 2-1. In all but one case, the correct choice is to follow the branch back and repeat the loop. This leads to a simple scheme of *branch prediction* take the same branch as the last time. Using this simple scheme, the loop branch will actually only cause one structural hazard. Another simple scheme is to take a branch if it leads back in the instruction stream, and not to take it if it leads forward. The latter approach favors loops and if-statements that evaluate to true.

Most modern processors use both of these heuristics to predict which branch to take. Branch prediction units today are implemented using branch history table indexed by the least significant bits of the address of the branch instruction. Each entry contains a couple of bits that indicate whether the branch should be taken or not, and whether the branch was taken the last time or not.

Maintaining a table like this is known as *dynamic branch prediction* and it works rather well in practice. RISC was designed with pipelining in mind, so they typically also allow for static branch prediction. Static prediction is done by having two kinds of conditional branch instructions, one for branches that are likely taken, and one for unlikely ones. It is then up to the compiler to figure out, which on should be used. In case of Algorithm 2-1, an intelligent compiler may realize that it is dealing with a loop that may iterate for a while, and generate a "likely conditional branch" instruction to do the loop.

Conditional Execution

There are cases where branches are simply not predictable, however. If *A* and *B* dereferences to random numbers, the branch inside the loop in Algorithm 2-1 is unpredictable, and we would expect a misprediction every other iteration. With a ten-stage pipeline, that amounts to an average of five wasted cycles each iteration, due to structural hazards alone. The only solution to this is not to use branches at all. The assignment in the branch inside the loop of Algorithm 2-1, could like this: `*u=*A*(*A<*B)+*B*!(*A<*B)`, assuming usual convention of Boolean expressions evaluating to 1, when true and 0 otherwise.

The expression is needlessly complicated, involving two floating-point multiplications; instead, a so-called predication bit is used. The test `*A<*B` is done, as when branches was used, and the result is put in a predication bit. Following the test comes two conditional move instructions, one to be completed only if the test was true, and the other only if the test was false. The result of the test is ready just before the first conditional move is to be executed, so no pipeline hazard arose. This way, instructions in both branches are fed through the pipeline, but only one has an effect. A primary limitation of the application of this approach is thus the complexity of what is to be executed conditionally.

Another limitation is that only simple conditions are supported, not arbitrary Boolean expressions or nested if-statements. Yet another is that typical architectures do not allow for all types of instructions to be executed conditionally, and for Algorithm 2-1 to be implemented without risk of severe hazards, we need both a conditional store

instruction and a conditional increment or add instruction[1]. Conditional instructions are native to RISC, but have also been added to the x86, from Pentium Pro and on, though only in the form of conditional moves.

2.2.2 Super-Scalar Out-of-Order Processors

The high price of branch mispredictions prevents us from gaining performance by making the pipeline longer. Orthogonal to parallelizing by making the pipeline longer, is parallelizing by issuing more instructions. Realizing that the execution unit often is the bottleneck in the pipeline, we could operate several execution units in parallel. An architecture that employs multiple execution units is known as super-scalar. Super-scalar architectures were popular in the 1970s and 1980s culminating in Very Long Instruction Word (VLIW) designs that packed several instructions into one. A single instruction in a VLIW architecture could consist, for example, of an integer multiplication, a floating-point addition and a load. Each of these instructions would be executed in its own unit, but they would be fetched and decoded as a single instruction. The interest in VLIW has diminished somewhat, probably due to the limited parallelism in real-life code, and limited ability of compilers to express it explicitly.

Going super-scalar with an architecture that was not designed for it, takes quit a bit of work. The 80486 processor was the first CISC processor, designed by Intel, to feature a pipeline. Its successor, the Pentium, expanded on that featuring two parallel pipelines, one (the u pipeline) capable of executing all x86 instructions the other (the v pipeline) only capable of executing a few select ones. If an instruction was immediately followed by one capable of being executed in the v pipeline, they could be executed simultaneously. If the following instruction could not be executed in the v pipeline, the instruction would simply go down the u pipeline alone the following clock cycle. Not surprisingly, only code generated specifically with this setup in mind benefit.

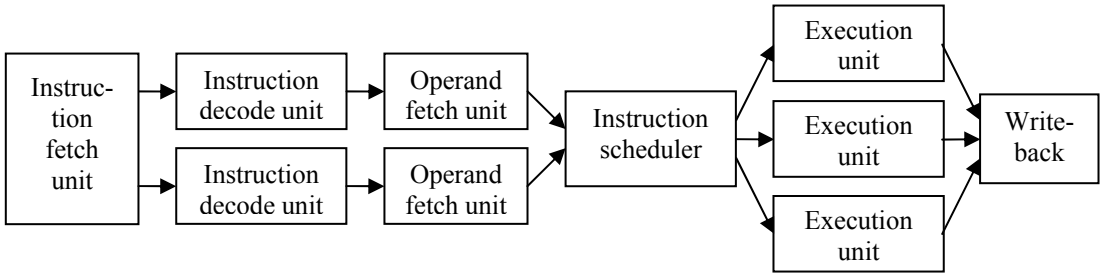


Figure 2-2. A multi-pipeline super-scalar processor.

The successor, the Pentium Pro, extended the number and capabilities of the execution units. It was capable of issuing several μ-ops per clock to the execution unit. Furthermore, the order of the individual instructions did not have to be maintained on entry into the execution units; it would be reestablished after the instructions had been executed. As long as one instruction did not depend on the result of one scheduled close before it, it would be executed in parallel with other instructions. A further advantage

[1] Conditional adds can be simulated with an unconditional add followed by a conditional and then an unconditional move, however that may be to much work.

of having multiple pipelines and execution units is that data and structural hazards not necessarily cause the entire processor to stall.

Register Renaming
It is actually possible to extract even more parallelism seemingly, from where no parallelism exists. Consider this C expression: `++j, i=2*i+1`. On an, admittedly contrived, one-register processor, the expression could be realized by the following instruction sequence:

```
load j    // load j into the register
inc       // increment the register
store j   // store register contents back in j
load i    // load i into the register
shl 1     // multiply by 2
inc       // add 1
store i   // and store result back in i
```

The first three instructions are inherently sequential, as are the last four. With only one register, the two blocks of code would step on each other's toes, were they to be executed in parallel. An out-of-order processor could execute the last four before the first three, but with no performance gain. If, however, the architecture had two registers, the first three instructions could then use one and the last four use the other, thus allowing them to be executed in parallel in almost half the time. This is exactly what register renaming does. Even though the architecture only allows the instructions to use one register, behind the scene instructions are mapped to different physical registers to extract parallelism. A so-called scoreboard keeps track of which scheduled instruction reads and writes which register, and allocates rename registers and execution units accordingly.

This approach works best, when the instruction set does not offer direct access to many registers. One may ask, why not simply add more general purpose registers, and let the compiler use them as best possible? The answer is simple: it will not make legacy code, compiled for fewer registers, run any faster.

2.2.3 State of the Art

The x86 architecture is still very much alive. It was extended with support for 64-bit addressing just last month. The Opteron, the first implementation with this new extension, has a 10K branch history table, three parallel decoder units, three address generating units (AGUs) and three arithmetic/logic units, as well as floating-point addition, a floating-point multiplication and a third float unit. Each pair of address generating and arithmetic/logic has an eight-entry instruction scheduler and the floating-point units have a 36-entry scheduler. Further, it doubles the number of general-purpose registers to 16 that code compiled specifically for the Opteron will be able to utilize.

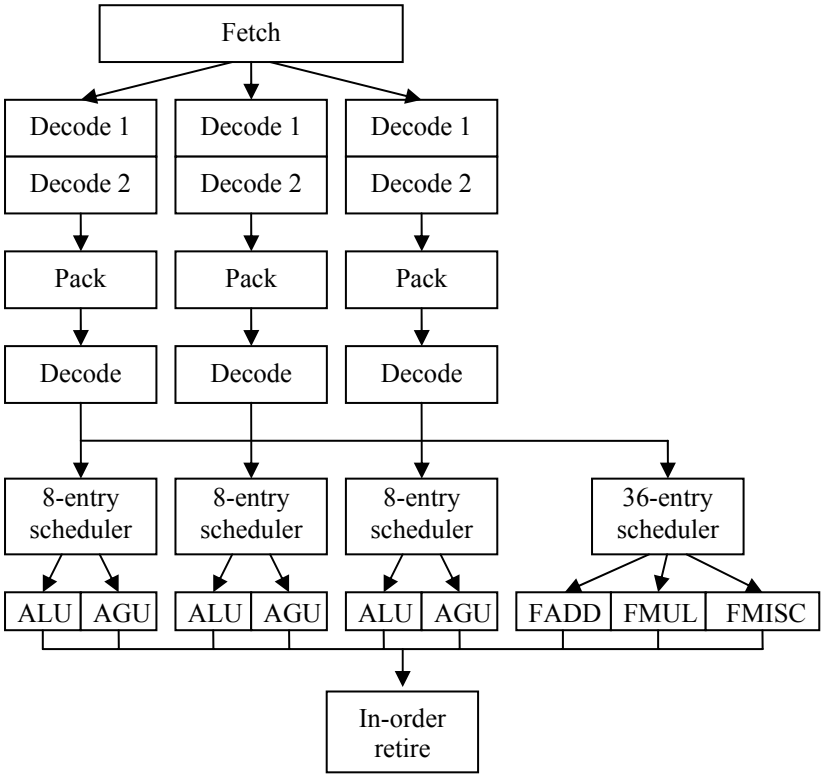


Figure 2-3. Core of the Opteron processor. The many decoder units are primarily due to the complexity of the instruction set.

Top of the line workstation processors from Intel today has a 20-stage pipeline, ALUs that work at twice the clock speed of the rest of the core, 128 rename registers (16 times the number directly available to the programmer/compiler). Their out-of-order logic can handle up to 126 μ-ops at a time. To minimize the risk of control hazards, they have 4000-entry branch history table and a 16-entry return address stack. RISC processors typically do not go to these extremes, but do also employ the types of optimizations described above.

2.3 Modern Memory Subsystems

With the introduction of DRAM and SRAM, a fundamental gap was established; DRAM chips can be manufactured using a single transistor per stored bit, while SRAM chips required six. DRAM chips however requires frequent refreshing, making it inherently slower. The question thus arose, should the computer have fast but expensive, thus limited in quantity, storage, or should it have cheap, slow storage and lots of it?

The question was easily answered up through the 1980s; CPUs were not much faster than DRAM chips so they were never starved of data, even if they used DRAM as primary storage. However, improvements in the transistor manufacturing process have since then made CPUs and SRAM chips a lot faster than DRAM chips. This, combined with the employment of super-scalar pipelined designs, have given CPUs and SRAM a

factor of 25 speed increase over DRAM chips in clock speed alone, with a further factor of three or four from synchronization, setup, and protocol overhead. Feeding the processor directly from DRAM, could thus make data hazards stall the pipeline for more than 100 cycles, significantly more than the number of cycles wasted on e.g. a pipeline flush.

Performance enhancing techniques, such as those discussed in the previous section, can also be applied to storage. However, when dealing with memory in a request-response pattern, *latency* aside from mere throughput, is an important issue. Latency is the time it takes a memory request to be served. Introducing pipelines or parallel access banks in RAM, enabling the handling of several memory requests simultaneously, may increase the overall throughput, but it will also increase latency.

2.3.1 Low-level Caches

The problem is economical rather than technological; we know how to build fast storage, but we do not want to pay for it in the quantities we need. The solution is to use hybrid storage, that is, *both* SRAM and DRAM. The quantity of each is determined solely economically, but spending about as much money on SRAM as on DRAM seem to be a good stating point. That, in turn, will give a lot less SRAM than DRAM.

The now widely used technique for extending storage with small amounts of faster storage, is known as caching. The goal is to limit the number of times access to slow storage is requested and the idea is to use the small amount of SRAM to store the parts of data in DRAM that is requested. When a processor needs a word from memory, it does not get it from DRAM, but rather from the cache of SRAM. If the data was not in cache, it is copied from DRAM to cache, and the processor gets it from there. If it is, however, we can enjoy the full performance of SRAM. The hope is that most of the time, we only need the data stored in fast storage, the cache. The challenge is then to keep the needed data in cache and all else in RAM, so we rarely need to copy data from RAM to cache.

Design

The design of the system managing the cache should be as simple as possible, to provide for high responsiveness; however, a too naïve design may cache the wrong data. One important observation to make is that data, whose addresses are close, tend to be accessed closely in time. This is known as *the locality principle* [Tan99, p. 66], [HP96, p. 373]. One example is the execution stack. Stack variables are rarely far from each other. Another is a linear search through a portion of memory; after having accessed the i'th word, we will in the near future access the $i+1^{st}$ through, say, $i+16^{th}$ word, which are all stored close to each other.

A simple design that performs well in these scenarios consists of splitting memory into *lines*, each line consisting of several, say B, words of data. When a needed word is not in cache, the entire line it resides on is brought in. This eases the burden on the cache management system, in that it only needs to deal with lines, not single words. This design provides for good performance on directional locality, such as linear searches, where after a program has accessed a word at address i, it will access the word at address $i+d$. If the access to address i caused a new line to be brought in (caused a *cache miss*), the previous access, to address $i-d$, could not have been to the same line. The word at address i must thus be at one end of a line, regardless of whether d is

negative or positive, so the next $B/|d|$ accesses must thus be within the line just brought in. The effect of this is that the time it takes to bring in the line is effectively amortized over the $B/|d|$ subsequent accesses. With d much smaller than B, this effect can be significant. Conversely, if d is larger than B, we do not gain from using a cache. By the way, it was the effect of a large d that caused Algorithm 1-1 to perform poorly, when the matrices became big. The effect splitting memory into lines effectively amortize the cost of accessing slow DRAM over the next $B/|d|$ accesses.

A system is needed to decide which line to overwrite, when a new line comes into cache. The ideal is to put it where the line, we are least like to use again it at. However, allowing lines to be placed at arbitrary positions in cache make them hard to locate, which in turn, reduces responsiveness. On the other hand, we are reluctant to let hardware requirements dictate where to put the data, since it may overwrite data we will be using soon. A more general approach is to divide the cache into *sets* of lines; a line may be placed arbitrarily within a set, but which set it must be put in, is dictated by a fixed mapping function, usually the least significant bits in the address. The structure can be seen in Figure 2-4.

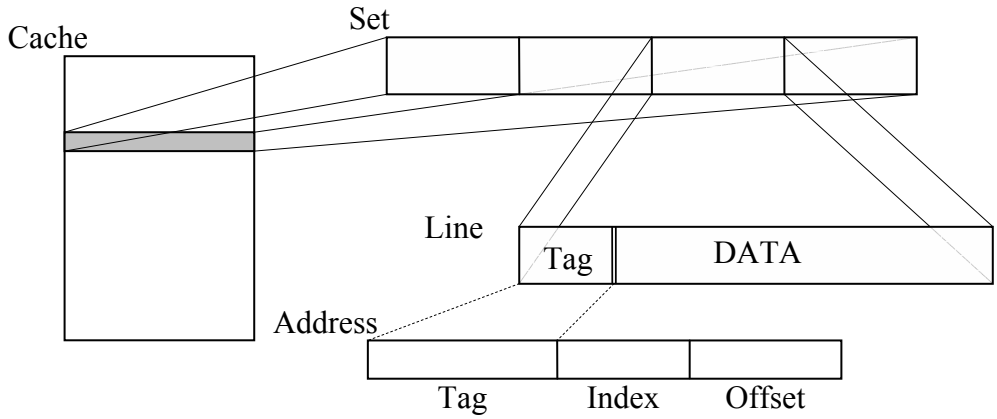


Figure 2-4. A four-way set associative cache.

When a word is needed, the index part of the address is used to locate the set. This is done quickly using a multiplexer. The tag part of the address is then checked against entries in the set. The tag of entries indicates exactly which line of memory is stored there. This comparison can be done in parallel with all entries in the set, however the more lines in a set, the more time is needed for the signal to propagate through. If a matching tag is found the word is in that line at an offset indicated be the least significant bits of the address. If it is not, however, the line is brought in from memory to replace one of the lines in the set. Which one to be replaced, is decided on an approximate *least recently used* (LRU) measure. The case of the cache consisting entirely of one set, known as a *fully associative* cache, allows lines to be placed at arbitrary positions in cache. Conversely, in *direct mapped* caches, a line can be placed only at the position dictated by the index part of its address in memory.

Miss Categories

It is sometimes useful to distinguish the different reasons for cache misses, so we may understand how to reduce their number. [HP96, Sect. 5.3] divides cache misses into three categories:

- *Compulsory*. The very first access to a word. By definition, the word will not be in cache and hence needs to be loaded.
- *Capacity*. If so much data is repeatedly accessed that the cache cannot contain it all, some lines have to be evicted, to be reloaded again later.
- *Conflict*. A variant of a capacity miss, occurring within a set; if data is accessed on lines that all map to the same set, a line from the set will have to be evicted and later retrieved.

Compulsory misses are usually unavoidable. Increasing the capacity to reduce capacity misses will also increase response time. To avoid this, the caching technique is often reapplied to itself, resulting in multiple levels of cache. The cache at the level closest to the CPU, named L1, is often designed as an intricate part of the core itself. It has very small capacity, but is highly responsive. The next level cache, L2, is significantly larger, but the overhead of locating data in it may be several more cycles. With this design, capacity misses only result in access to DRAM when the capacity of the L2 cache is exceeded and the L2 is checked only if a word is not in L1 cache. The L2 cache is often placed on the same chip as, but not embedded in the core of, the CPU. High-end processors may employ a third level of cache, sometimes shared with other processors in multiprocessor environments.

One consequence of using multiple processors is that cache contents can be changed externally. To check if the cache contents are being modified from another processor, a processor monitors the write requests on the memory bus (called *snooping*), and checks each of them to see if it affects data in its cache. This checking can interfere with normal cache operations. To avoid this, the contents of the L1, complete with data and tags, are often duplicated in L2. This way, the snooping mechanism can check write requests against the tags in L2 in parallel with the processor working on data in L1. A cache designed this way is said to have the *inclusion property* [HP96, p. 660-663,723-724]. As always, there is a tradeoff. Applying the inclusion property wastes space in the L2 cache, so depending on the relative sizes of the two levels, it may not be efficient. For example, the Duron, Athlon, and Opteron processors from Advanced Micro Devices (AMD) all have a total of 128kB L1 cache, making it inefficient to duplicate its contents in L2 cache. While AMD does not apply the inclusion property, most others do.

Conflict misses are strongly related to the degree of associativity in the cache. A fully associative cache will not incur conflict misses, while direct mapped caches have a significant likelihood of incurring conflict misses. On a k-way set associative cache, a program must access data at no less than $k+1$ location, all with the same index part of the address and different tags, in order to incur capacity misses. With $k > 1$, this is reasonably unlikely, unless software is designed intentionally to do so. Paradoxically, software designed to minimize capacity misses tend inadvertently to increase accesses to data within the same set (see Section 6.1.2 or [XZK00]). Using direct mapped caches completely eliminates the need for tag matching and LRU approximation hardware, thus reducing the complexity significantly. Hence, direct mapped caches has been the

design of choice among RISC designers, however, lately the risk of conflict misses has pushed RISC to adopt multi-way set associative caches. Now only very few processors have direct mapped caches and those that do have multi-way associative L2 caches. Among CISC, the Pentium 4 has a four-way set associative L1 cache and eight-way L2 while the latest x86 incarnation, the Opteron, has a 16-way set associative L2 cache.

Recall the structural pipeline hazard occurred when the fetch stage attempted to read an instruction from memory at the same time the execution stage read data from memory. Having one cache dedicated to instructions and one for data virtually eliminates the risk of structural hazards due to the fetch stage accessing memory at the same time as the execution unit, since the fetch and execute units each access their own cache. L1 cache is often split up this way, while L2 contains both instructions and data. This way, many instructions can be kept close to the CPU and the risk of both the execution and the fetch stage causing an L1 cache miss and accessing L2 in the same cycle, thus incurring a structural hazard, is virtually gone. Another reason for having separate instruction and data cache is that the access patterns for instructions tend to be more predictable than for data. The instruction cache can exploit this by e.g. load more than one line at a time.

2.3.2 Virtual Memory

Storage limitations not only had an influence on the central processing units; programmers tend in general to need more space than can be afforded. This is as much the case today as it is back in the middle of the last century. Indeed, the idea of using a memory hierarchy to cheaply provide a virtually infinite storage can be traced back to von Neumann in 1946 [HP96, Chap. 5], before even magnetic core memory was invented.

In the early 1960s, a group of researchers out of Manchester designed a system, known as virtual memory that provided a significant boost in storage capacity without needing the programmer to do anything. The idea was to work with the disk, usually considered secondary storage, as primary storage, but with random-access memory as a cache. Its implementation had very little hardware requirements, in fact, what they needed, they build themselves. It was designed primarily to ease the burden of managing the overlays used, when programs became too big to fit in memory and as an added benefit, it eased the use of large data sets too [Tan99, Chap. 6].

It has had at least two major impacts on the way programs are developed today. Firstly, programmers do not have to consider how much physical memory is present in the computer, the program is running on; a program designed to run on top of a virtual memory system will run on a computer with a gigabyte of memory as well as on a computer (telephone?) with only 16MB on memory. Secondly, for most applications, programmers do not need to consider the case of running out of memory.

Paging

The approach to implementing virtual memory is the same as implementing caches; the storage is split into blocks of words that are moved to and from cache atomically. In virtual memory terminology, these blocks are not called lines but *pages* and slots where pages can be stored in memory are called *page frames*. A process is allowed access to any address, not just the ones that fall within physical memory. Addresses used by the process are called *virtual addresses* and are taken from the *address space* of the process.

They are translated into the address in physical memory containing the desired word by the *memory management unit* in the processor.

The virtual to physical address translation is done using *page tables*. These tables are indexed, rather like the sets of the L1 and L2 caches, by the high order bits of the virtual address; however, unlike the low-level caches they do not contain the actual data. Rather, each *page table entry* contains the physical address of the page frame, where the page is located. There are no restrictions on what frame an entry must use, thus paging is fully associative.

When a virtual address cannot be translated to a physical, e.g. if the page is not in physical memory, the processor generates an interrupt, called a *page fault*; the processor it self is not responsible for bringing the page into memory. Since disk management is complicated and is highly dependant of the system used to manage files on disk, this is best left to the operating system. Unlike with lower-level caches the latency of the storage being cached (disk) is much higher than that of the cache, so we can afford to let software handle a miss.

Each process has its own address space and hence its own page table. To service a page fault, the operating system determines which process is faulting and which virtual address caused the fault. It then examines the page table of the process and locates the entry. When a page is in memory, the entry points to the frame containing it, but when it is not, the operating system is free to use it to point to a specific position in a specified file on disk. Using this information, the operating system reads the page from disk, stores it in a page frame, updates the page table accordingly, and lets the process continue running.

The maximum size of the address space of a process is dictated by the number of bits in the operands of address operations. A 32-bit processor thus supports processes using 2^{32}B = 4GB of virtual memory. To avoid flushing caches and the TLB (se below) on system calls, half of the address space is often mapped to kernel data structures. If the page size is 4kB, each process would need a page table of one million entries, each of about 32 bits, for a total of 4MB. While the tables are data structures stored in the virtual memory of the kernel, and thus may be paged, 4MB is still a lot of space. One solution is to use hierarchical page tables. A *page directory* of a thousand entries can point to page tables, each of a thousand entries mapping from 4MB of virtual memory to page frames. This way, if a process only uses 10MB of not too scattered storage, it would only require a page directory and three page tables. The downside is that two table entries need to be looked up per memory reference. The translation process can be seen in Figure 2-5.

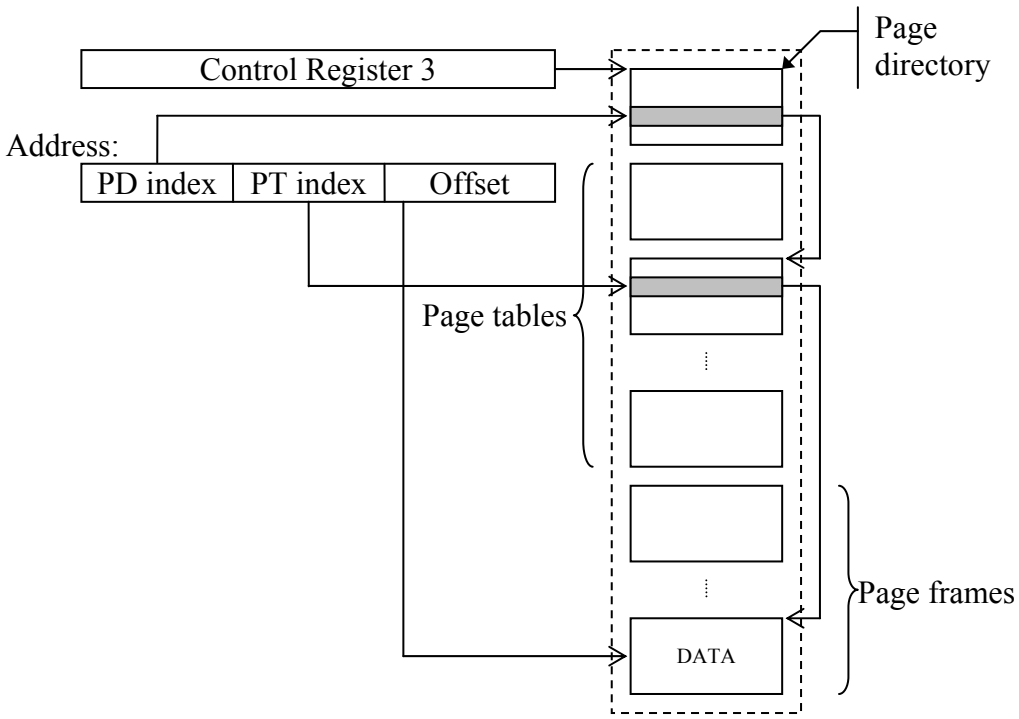


Figure 2-5. Virtual to physical address translation with two levels of page tables. A register in the processor (incidentally called CR3 in x86 terminology) locates the page directory.

Page Replacement Strategies

When a page is brought in from disk and memory is full, it needs to replace some other page in memory. Operating the disk to get the page typically takes in the order of milliseconds, while access to RAM takes in the order of tens of nanoseconds. Thus, we could execute millions of instructions in the same time it takes the page to get from disk to memory. One would think some of that time would be well spend contemplating exactly which page should be evicted, leading to very good page replacement strategies.

However, good replacement strategies, such as evicting the least recently used page, depend on information about pages (their *age*) being updated not only when they are evicted but also every time they are referenced, and we do not want to sacrifice any performance on all the references that do not cause page faults. To keep performance from suffering, non-faulting memory references must be handled entirely in hardware and making hardware update an LRU data structure would be infeasible. In fact, most hardware only keeps information on whether a page has been accessed and whether its contents have been changed in the *referenced bit* and the *dirty bit* respectively [Tan01, Sect. 4.3]. Therefore, while we could spend thousands, even millions of cycles finding the best page to evict, we only have two bits of information per page to base the decision on.

How these two bits are used, differ from operating system to operating system. A frequently used design includes a dedicated process, in UNIX called a *page daemon*, that scans pages and frees those deemed not needed. The hope is that the amount of free

space can be kept above some minimum, so that when a page fault occurs, we are not forced to deal with finding a page to evict. When the daemon frees a page, it is not necessarily evicted from memory rather it is put on a free list. Dirty pages on the free list are written to disk, so that when a page fault occurs, the incoming page can be placed in any of the free pages. If a page on the free list is needed, and not overwritten by an incoming page, it can be retrieved. This is known a minor page fault, as opposed to major page faults that involve disk access.

In version 2.2 of the Linux kernel, the page daemon does not do much other than scan and free pages. Its state consisted of the number of the page it inspected when it last went to sleep and it would start from there, scanning pages in memory and simply evict all pages that were not marked as having been referenced. All pages that had been referenced would have their referenced bit cleared, and if it is clear the next time the process comes by, it will get evicted [Tan99, Sect. 10.4]. This results in a mere *not recently used* (NRU) page replacement strategy. Version 2.4, like 2.0, uses an approach much as many other operating systems do, known as *aging*. The implementation requires the operating system to keep a table of the age of each allocated page in the system. When a page is first read in, its age is zero. Each time the page daemon passes it and its referenced bit is not set, its age is increased. If the referenced bit is set, its age is reset to zero. The choice of pages to free is now based on the age of the pages; the oldest ones are freed first, which yields a strategy closer resembling LRU replacement.

File Mapping

Hardware and software is now in place that allows the operating system to place disk-based data in memory, when a program accesses that memory. When a process allocates some memory, writes data in it, and it is paged out, the operating system knows where to get it, should the process need it again.

A slightly different take on this is to let the process specify where the data should come from, in case it is not in memory; it could specify a region in its address space and a file in the file system and when it accesses data in that region, the operating system reads in a corresponding page from the file. This way a program can read and write files without using standard input/output system calls such as `read`/`write`/`seek`. This feature is known as memory mapping of files and is provided by almost all operating systems today, indeed most operating systems use it to load in binary files to be executed.

Translation Look-aside

As can be seen in Figure 2-5, much work is involved in any memory reference, not only the ones that causes page faults. To access a word through its virtual address, aside from accessing the word it self, two additional memory accesses are needed to locate the word. Even though this is all done though hardware, given that DRAM is about two orders of magnitude slower than the CPU, this is a significant overhead. Indeed, the page tables them selves may be paged out, and so the accesses just to get the address of the word may cause page faults and disk access. This does not, however, happen frequently; a page full of page table entries covers (with the figures from above) 4MB of physical memory and any access to those 4MB would reset the age of the page containing the page table, making it unlikely it will get paged out, if it were in fact in use.

To aid in the complicated translation of virtual addresses, processors often provide an on-chip cache of recently translated addresses, called the *translation look-aside buffer* (TLB). Whenever a memory operation is executed, the TLB is checked to see if it falls within a page that has recently been translated. If it does, the physical address is looked up in the TLB and used to carry out the memory operation. If it is not, however, a *TLB miss* occurs, and the page tables are consulted with the overhead described above. As a cache, some TLBs are fully associative while others are two- or four-way set associative, though rarely direct mapped.

Memory operations can now be divided into three categories: a TLB hit, a TLB miss, and a page fault. As stated above, page faults should be handled by the operating system and a TLB hit should be handled with no overhead by hardware. However, whether a TLB miss should be handled by software or hardware is not clear; provided a good TLB implementation, TLB misses should be relatively rare and the added overhead of a software miss handler compared to the three hardware memory lookups may not have much of an impact.

The CISC approach is not to care about hardware complexity and thus, the IA32 clearly specifies the formats of the page directory and page tables, so that a TLB miss can be handled entirely in hardware. This will be the fastest way of handling a TLB miss, but also the most complex, costing transistor space that could be used for bigger TLBs or bigger caches. The MIPS RISC processor takes the opposite approach, where a TLB miss is handled much like a page fault. This way, the processor only needs to be concerned with looking up physical addresses in the TLB. When a TLB miss occurs on a memory request, the processor generates an interrupt and expects the operating system to provide a TLB entry so that request will not miss on the TLB. This provides for much greater flexibility in choosing the page table data structures. For example, the operating system can keep its own software TLB with much greater capacity, than the one in the memory management unit. The downside to use software TLB miss handlers is, of course, that it takes significantly longer to service the miss. Maybe we will soon see second level hardware TLBs caching those translations that cannot fit in first level.

2.3.3 Cache Similarities

Common for all levels of cache in modern computers, is that they consist of a number of blocks of words. Each block is transferred from one level to the next atomically. Say a cache has a capacity of M words and its blocks a capacity of B words. A process can repeatedly access B contiguous words incurring only one memory transfer and M contiguous words incurring roughly M/B memory transfers. The number of transfers is inversely proportional to the size of the block transferred, so in designing the cache it is tempting have large blocks. One negative side effect of this is that it will increase the transfer time and the price of a transfer must be paid if only one word is needed. Another is that it decreases the granularity of the cache. Assuming adequate associativity, a process can repeatedly access M words at M/B *distinct* locations, again incurring M/B transfers. This flexibility is a good thing and increasing B will reduce it. Never the less, high-end workstation and server operating systems favor larger page sizes to maximize throughput.

The above description extends to the translation look-aside buffer as well. Say the buffer has T entries. With $M = TB$, B being the size of a page, we can access B

contiguous words incurring at most one TLB miss and M contiguous words or words at up to $M/B = T$ distinct locations, incurring at most $M/B = T$ TLB misses.

2.4 Summary

Achieving high performance in modern computers today comes down to keeping the pipeline fed. Two major obstacles to achieving this are control and data hazards, caused by unpredictable branches and slowly reacting storage.

Unpredictable branches are expensive, especially in long pipeline designs of CISCs, while predictable ones need not be. They can cost between ten and twenty cycles, between five and ten on average. Conditional execution can be used to avoid branches. However, the limited set of instructions capable of being executed conditionally limits its usefulness. Conditional moves can be used in such problems as selecting a distinct element, say the smallest, from a set of elements. As pipelines get longer and longer, more precise branch prediction units are needed. Recent pipelines of 12+ stages are accompanied by very capable branch prediction units, so that unless branch targets are completely random, branches are executed almost for free. Indeed, if the branch test is independent of instructions executed prior, it can be evaluated in a separate execution unit in parallel with other instructions, and if the branch was predicted correctly, the branch does not interfere with execution of the other instructions. In Algorithm 2-1, for example, the evaluation of `++i` and `i!=n` can be executed in parallel with the loop body, so when the branch is (correctly) predicted taken, some execution units proceeds with the loop body while one execution unit increments `i` and verifies that `i!=n`, while the loop body is executing at full speed.

Non-local storage accesses may take very long time. Requests go through several layers of cache, and the further they have to go, the longer the pipeline has to wait. If the requested data is in L1 cache, the pipeline is usually stalled for no more than one cycle and for no more than 5-10 cycles, if it is in L2. If it is not in cache, the delay depends on whether the address can be easily translated through the TLB. If it can, the delay may be as low as 100 cycles; otherwise, it will be two-three times as much or in case of software TLB miss handlers, significantly more. If a page fault occurs, the process will likely be stalled for tens millions of cycles.

Managing the cache needs to be fast, so no level provides exact least recently used block replacement, but all provide an approximation. The increased complexity of multi-way set associative caches has lead RISC processor designers to stick with direct-mapped caches for a long time; only the very latest generations of RISC processors, such as the UltraSparc III, the PowerPC 4, and the MIPS R10000 use multi-way set associative caches. The Pentium 4 has an eight-way set associative L2 cache and even a four-way set associative L1 cache and the Opteron has 16-way set associative L2 cache. With this increased flexibility, conflict misses should become less predominant.

On the positive side, if there is any parallelism in the code, we can expect the processor to detect and exploit it. Executing code like `i=i+1, j=2*j+1` may thus not be any more expensive than merely incrementing `i`, simply because there is no dependency between instructions updating `i` and the instructions updating `j`, so they can be scheduled in different execution units.

Chapter 3

Theory of IO Efficiency

Having now realized that instruction count is not always a good indication of the running time of an algorithm, we will be interested in obtaining a theoretical understanding of what might then influence it.

In this chapter, formal models capturing the effects of the cache-misses and page faults are introduced. Lower bounds on the complexity of sorting are proven and a cache-aware algorithm meeting that bound is given.

3.1 Models

Theoretical algorithm analysis has been founded on one fundamental activity: counting the number of instructions executed. Knuth pioneered this discipline in the work The Art of Computer Programming [Knu98]. The approach was to develop a hypothetical, yet then representative instruction set, dubbed MIX, then implement every algorithm in the book using this set and thoroughly analyze the number of instructions executed in each of them. The result was very precise statements on worst- and average-case instruction count of each algorithm.

Today, this attention to detail is not often seen; a high-level description of an algorithm is preferred, to ease understanding and to limit cluttering with instruction set specific details. In addition, when implementing algorithms we do so in high-level programming languages for portability and genericity. Exactly which instructions are hidden behind the high-level language constructs is not important for the analysis; a focus on only a subset of the instructions has been prominent. For sorting, the subset has traditionally been a comparison instruction, while numerical computations have been a particular floating-point operation, say multiplication. These instructions can easily be isolated, even if the algorithm is only described in a high-level language. To ease the burden of analyzing algorithms further, we use asymptotic notation, which allows us to discount low-order terms and constants in high-order terms.

3.1.1 The RAM Revisited

The strength of a model lies in part in its ability to support lower bounds on the complexity of interesting problems; without lower bounds, we cannot prove optimality of algorithms solving the problems.

Consider sorting in the RAM model. An easier problem than sorting is determining the permutation that will bring the elements in non-descending order. The number of permutations of n elements is $n!$. When doing a one comparison, the order of two

elements is determined. This order can be chosen so at most half the remaining permutations are excluded. It thus takes at least $\log(n!)$ comparisons to exclude all but the right permutation. Recall, that in the RAM model, we are not allowed to examine the individual bits of the elements, so the only way of excluding permutations is by comparing elements. The argument is then that a correct sorting algorithm must be able to exclude all but the one permutation that will bring the elements in this order. Say it had excluded all but two permutations π_1 and π_2, and it simply picked one, say π_1, rather than do the last comparison the find the correct one. Then there would be an input to the algorithm that it would not be able put in correct order, namely the permutation π_2^{-1} of a sorted sequence. The argument that there exist inputs, on which the algorithm will be incorrect, is known as an *adversary argument*; an adversary decides the input, or rather answers queries about it in a consistent manner, and if it is possible to do so in a way that reveals flaws in the algorithm, it cannot yet have solved the problem. A similar, simpler argument shows that there exists an input to a correct sorting algorithm, on which the algorithm must make $n+1$ moves, assuming output must reside in the same locations as the input.

In the RAM model, all operations take unit time so a sorting algorithm must take at least $\Omega(\log(n!))$ units of time, even if it also only does n moves. Consider such an optimal sorting algorithm. On any given hardware implementation (and a given type of elements), there exist a constant $c > 0$, such that it takes c times longer to move elements, than to compare them. Discounting other operations, the running time of such an algorithm will be proportional to $\log(n!)+cn$. Using asymptotic notation, we would ignore the latter term, but what happens, when constants matter?

A floating-point division may for example take 25 times that of a multiplication, which takes five times that of integer addition. If an algorithm makes n divisions and $n\log n$ additions, we say that the complexity is $\mathcal{O}(n\log n)$, but when $\log n < 125$, that is, for all $n < 10^{37}$, the execution time will be dominated by the time it takes to perform the divisions, which is $\mathcal{O}(n)$. In the previous chapter, we saw that not even identical instructions execute in the same amount of time; an instruction executed in one context may execute in twenty-thirty, maybe even a million times the time in which the same instruction executed in a different context. We saw certain memory operations were a very large factor slower than any other operation.

Overzealous use of asymptotic notation hides important aspects of algorithm performance; however, the solution is not to abandon asymptotic notation, but rather the model. In the RAM model, memory operations are just another type of operation. This observation invites the idea, that in estimating running time of an algorithm, we should not count the simple operations, rather these types of memory operations.

3.1.2 The External Memory Model

In 1972, Floyd pioneered the notion of analyzing the number of transfers between primary and secondary storage, proving upper and lower bound on transfers incurred during matrix transposition [Flo72]. Nine years later, Hong and Kuhn derived a lower bound for Fast Fourier Transformation [HK81]. The next major step was taken in [AV88], when these results were proven in a more general setting: The *external memory model*. Specifically, what this new model was able to account for was that elements were transferred in blocks with a non-constant capacity that is independent of memory size. A lot of work has since been done, both practical and theoretical, in

developing efficient algorithms in this model (see [Vit01] for survey). A multitude of other models, most more complex than the external memory model, such as the multi-level memory model, the hierarchical memory model, and the uniform memory model, that attempts to model the memory hierarchy has also since been proposed (see [FLPR99, Sect. 7] for an overview and references).

The external memory model is a model of secondary storage, rather than of computation. In the external memory model, secondary storage consists of a random-access magnetic disk. All computation is done in *and only* in primary storage (memory), and data is transferred from secondary storage (and back) in *blocks*. The model is characterized by these parameters:

- Problem size N: the number of elements to be sorted.
- Memory size M: number of elements that can fit in memory.
- Block size B: number of elements that can be transferred in a single block.
- Number of disks P: number of blocks that can be transferred concurrently.

The following relation is said to hold: $1 \leq B \leq M$. For the discussion of the external memory model in this thesis, we will concentrate on the case $P = 1$. Note that these parameters are given by the concrete implementation of the model, i.e. the machine on which the algorithm is run. Figure 3-1 depicts such a machine.

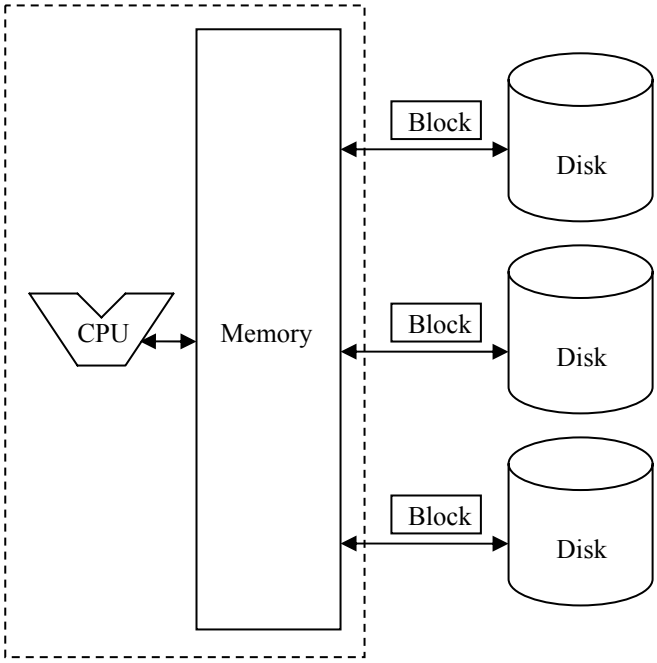


Figure 3-1. Illustration of the external memory model with $P = 3$ parallel disks.

The input of an algorithm in the external memory model is initially placed on disk, from where it can be read into memory in blocks (see Figure 3-1). There can be no more than M/B blocks in memory at any one time, so blocks may also have to be written back to disk, to not loose data; if a block is read into and the memory is already filled, some portion of the memory will be overwritten. If it matters, for correctness or

complexity, it should be stated exactly where the block should be placed in memory; the model states that memory is fully associative, so we are free to choose where is should go. When blocks have been written to disk, they do not occupy space in memory. The complexity of the algorithm is measured in the number of times it transferred a block from or to disk. A transfer is also known as an I/O, which is shorthand for input or output.

Since the model is essentially that of a two-level storage system, where data is transferred in blocks, it may also be used to describe other two levels of the memory hierarchy, such as L2 cache and DRAM memory or TLB and page tables, the value of constants M and B are merely different. Note, in practice we have neither control of when the transfer of blocks should take place, nor where to put the blocks in cache. Indeed most L2 caches are not fully associative.

A Simple Example: Scanning

Consider the simple problem of computing an aggregate of N elements, for example the largest element or the sum of the elements. Algorithm 3-1 solves the problem in the external memory model. It assumes the elements are stored contiguously on disk, so that the i'th block contains elements $(i\text{-}1)N/B$ through $iN/B\text{-}1$. For simplicity, we assume $N = cB$, for some positive integer c.

Algorithm 3-1. EM_sum

```
sum = 0;
for( int i=0; i!=N/B; ++i )
{
    read the i'th block into memory;
    compute the sum s of elements in the block;
    sum += s;
}
```

We use the high-level description compute the sum s of elements in the block, because exactly how it is done does not matter in the analysis, and so we leave out the details. It is easy to see the algorithm is correct (provided s is computed correctly). The analysis is equally simple: the algorithm performs N/B reads. This is optimal because computation can only be done in memory and as with all aggregate problems, if an algorithm ignores one or more elements of the input then the algorithm will be incorrect on inputs, in which the aggregate depends on the ignored element, so all elements have to, at some point, have been in memory. We transfer elements in blocks of B elements, so at most B elements can get into memory per read. Since at least N elements have to be read in, a correct aggregate computing algorithm must perform at least N/B reads.

The technique used in Algorithm 3-1 is simple, yet it illustrates some important points in dealing with I/Os. The placement of the input on disk is essential; were the elements scattered randomly around the disk, we would not be able to get B elements into memory in a single read. The region of memory where the block read in is stored, is called a *buffer*. When laid out contiguously, we were able to touch elements at an amortized cost of B^{-1} reads per element. We will call a collection of elements with this property a *stream*. Streams are often either input or output streams, where output streams can store a collection of elements on disk at an amortized cost of B^{-1} writes per

element. This is done by collecting elements in the buffer, writing it to disk only when it becomes full. Note that if we read in the next block to the place in memory the previous block was stored, the algorithm would still be correct. Thus, a stream requires no more memory than one block occupies.

The third point is specific to the model; the values of M and B (and P) are specific to a concrete instance of the model. When implementing e.g. Algorithm 3-1, we would either have to decide on a value for B, in which case it would be suboptimal when run a machine with block size $B' \neq B$, or we would have to figure out what B to use at runtime, information that is not in general available. Not having access to the value of M and B in an implementation implies, that either the implementation is not optimal, or they will become suboptimal, when the values change e.g., when the memory of a computer is upgraded.

Binary search

Another simple and illustrative problem is that of finding a particular element among sorted set of elements. In the RAM model, this can be done efficiently using a balanced binary search tree in which elements are stored in the nodes. Elements stored in the left subtree of a node are all smaller than the element in the node and all elements in the right subtree are greater. This property is used to navigate down through the tree to isolate the desired element, in time proportional to the height of the tree, which is $\mathcal{O}(\log(n))$. That this is optimal can be realized by noting that each comparison can be chosen by the adversary to reduce the set of candidate keys by at most a factor of two. After $\Omega(\log(n))$ comparisons, we have thus eliminated all but the right key.

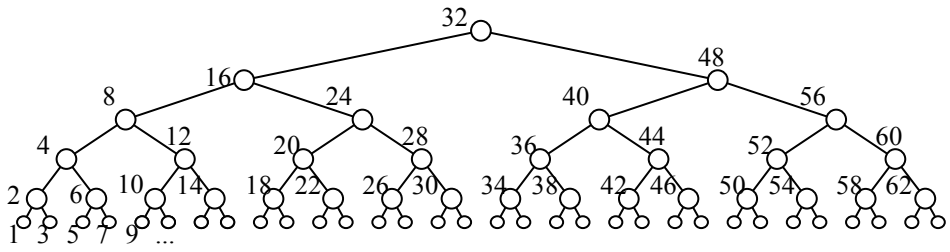


Figure 3-2. A binary search tree. The number by the nodes indicates the rank of the element stored there.

In the external memory model, reading a block for each new level of a binary tree will be suboptimal. Instead, we use B-trees, where each node contains a block of elements and has $B+1$ subtrees. The elements in the block functions as partitioning elements for the elements in the subtrees; all elements in the i left most subtrees of a node are smaller than the first i elements of the block in the node. When doing a search the block associated with a node is read in, using one I/O. Using the elements in the block, the node in which to continue the search is determined. This way we still do one I/O per level in the tree, but the height of the tree is only $\mathcal{O}(\log_B N)$, and thus so is the complexity. This too is optimal by the same argument as above, except an I/O now read in B elements and the adversary can only choose an outcome that reduces the set of candidate keys by a factor of at most B.

3.1.3 The Cache-Oblivious Model

As discussed in Chapter 2, well-established ways of managing the reads and writes from and to disk, so that the programmer need not know the details of disk input/output, already exist namely using virtual memory. Programmers have become used to the same abstraction on all levels of the memory hierarchy. The *cache-oblivious model*, introduced by Frigo, Leierson, Prokop, and Ramachandran in 1999 captures this way of abstracting memory transfers away from algorithms and their implementations [FLPR99].

The idea of the cache-oblivious model is strikingly simple; design the algorithm for the RAM model, but analyze it in the external memory model. Perhaps the most profound consequence of this is, since the algorithm is oblivious to the structure of the memory hierarch, an optimal cache-oblivious algorithm is *automatically optimal on all levels of the memory hierarchy, and on all hardware implementations of these hierarchies*. We now no longer refer specifically to memory and disk, so in this thesis, when discussing cache-oblivious algorithms, we will use the term *memory* for general storage, *cache* for the faster level of storage that, caches elements stored in memory, and a *memory transfer* to refer to the process of moving a *block* from one level to another. The complexity of an algorithm in the cache-oblivious model is both the work done in the RAM model and the number of memory transfers.

Stating that an algorithm, described for the RAM model, is optimal on any level in a memory hierarchy when analyzed in the external memory model obviously requires some assumptions. Aside from assuming the levels work like the external memory model, in that the first level is fully associative and inclusive, and elements are transferred in blocks, we need an assumption on the page replacement strategy. These assumptions transform the external memory model to the *ideal cache model*, in which algorithms are then analyzed. The assumptions made in the ideal cache model are:

- *Optimal replacement*: When blocks are transferred to memory, the underlying replacement policy is the optimal offline algorithm.
- *Exactly two levels of memory*: There are no more than two levels of memory, that is, more than one level of cache. This is not a restriction compared to the external memory model, but it is compared to other, more sophisticated models like multi-level models.
- *Automatic replacement*: Blocks are transferred automatically, not explicitly by the algorithm. This is in contrast to models like the external memory model, where memory management is done explicitly.
- *Full associativity*: Blocks can be placed anywhere in memory.
- *The cache is tall*: See details below.

With the introduction of the ideal cache model, detailed proofs were given that algorithms that are optimal in the ideal cache model are also optimal in other models, such as the external memory model, where e.g. automatic replacement is not present, and multi-level models [FLPR99]. The computers, that are the targets of the engineering effort of this thesis (modern computers as discussed in Chapter 2), naturally fulfill most of these assumptions. Specifically, will we use the convenient virtual memory abstraction, so that automatic replacement and full associativity is a given. As for limited associativity in lower level caches, the construction used in the justification argument given in [FLPR99] is not of practical use due to a large overhead,

so we would have to find another way of dealing with limited associativity anyway. We will analyze the algorithms in a two-level memory model, and prove they are optimal, between these two levels, regardless of the architectural parameters. This way we prove they are optimal between any two levels but not that they are optimal on all levels simultaneously. That they indeed are is also proven in [FLPR99], but using an assumption that all levels of cache are inclusive, which is not always the case, specifically not at the levels of virtual memory.

Optimal Replacement Assumption

One assumption no computer will ever fulfill, however, is that of optimal replacement. Since cache-oblivious algorithms are unaware of the underlying caching mechanisms, they are unable to control them. Elements are transferred between levels in blocks by an underlying mechanism; if a particular element is not in cache, it must be brought in. Now, the mechanism must decide where to put it. In the ideal cache model, the choice is simply to pick the optimal replacement. This eases the analysis, in that when arguing upper bound on the complexity, we may simply say that the page replacement mechanism chose whatever we needed it to choose; if it did not, it must have chosen something even better, since it is optimal, thus improving the complexity of the algorithm. However, is it too unrealistic? Most real life caching mechanisms use LRU, or some more or less crude approximation.

The argument for optimal replacement being a reasonable assumption is based on a result by Sleator and Tarjan. It states that the number of cache misses Q_{LRU} on a cache using LRU is $M_{\text{LRU}}/(M_{\text{LRU}}\text{-}M_{\text{OPT}}+1)$-competitive with an optimal replacement strategy [ST85], that is

$$Q_{\text{LRU}} \leq \frac{M_{\text{LRU}}}{M_{\text{LRU}} - M_{\text{OPT}} + 1} Q_{\text{OPT}} \tag{3.1}$$

with Q_{OPT} being the number of cache misses incurred by a sequence of memory requests with an optimal replacement strategy, and M_{LRU} and M_{OPT} being the size of the caches for the LRU and optimal strategies respectively. An algorithm incurring $Q(N,M/\gamma,B)$ memory transfers on an optimal replacement cache of size M/γ, for some constant $\gamma > 1$, will thus incur no more than $\gamma M/(\gamma M\text{-}M+\gamma)\,Q(N,M/\gamma,B) \leq \gamma/(\gamma\text{-}1)\,Q(N,M/\gamma,B)$ transfers on an LRU cache of size M. So if $\gamma Q(N,M/\gamma,B) = \mathcal{O}(Q(N,M,B))$, which is known as the *regularity condition*, the number of transfers on an LRU cache, $\gamma/(\gamma\text{-}1)\,Q(N,M/\gamma,B)$, is $\mathcal{O}(Q(N,M,B))$, which was the number of transfers incurred according to the analysis in the ideal cache model.

Tall Cache Assumption

When solving non-trivial problems, it is common to exploit a certain level of granularity in the cache; put simply we would like the cache divided into more blocks than there can be elements in one block. To achieve I/O optimality cache-obliviously, we thus assume that there exist a (positive) constant c, such that $M/B \geq cB^{2/(d\text{-}1)}$, with the value of d being specific to the implementation of the algorithm, though strictly larger than 1. In case of the problem of sorting, this assumption has been proved both necessary ([BF03]) and sufficient (Chapter 4).

In theory, the ideal cache model is justifiable *in asymptopia*, but it is still a major open question, whether the performance of cache-oblivious algorithms is influenced by the less than ideal memory systems of real life. In particular, when faced with a direct-mapped cache we cannot, in practice, build up the simulation based on 2-universal hashing suggested by Prokop *et al.*, due to the very high overhead. Likewise, the tall cache assumption is in some sense always true, since we are free to choose c, however being forced to choose c small to fulfill the tall cache assumption, will impact the performance in a way, that is hidden in the asymptotic analysis.

None of the assumptions made by the ideal cache model avoids the basic need for blocks being transferred from level to level; computation can only be done in cache, and the only way to bring data there is by block transfer. This in turn means that the lower bounds that hold in the external model also hold in for the I/O complexity of cache-oblivious algorithms. Likewise, the lower bounds of the RAM model also hold for the work done by a cache-oblivious algorithm.

Scanning Revisited

Let us return to the simple aggregate computation example from the external memory model and see how it looks in the cache-oblivious model. In the cache-oblivious model, we do not control the disk, and so may seemingly not be able to control the layout of elements in memory, in a way that was so crucial to the performance of Algorithm 3-1.

Instead, we use the *array* construct. Arrays are collections of elements that are placed contiguously in the address space. On all levels of cache, a block of elements contains B elements that are contiguous in the address space, so accessing B elements that are contiguous in the address space, will cause no more than two blocks to be transferred, indeed accessing N elements contiguous in the address space cause no more than $N/B+2$ memory transfers. This is exactly what splitting up memory in blocks is designed to be efficient at. While arrays may not guarantee that elements are placed contiguously in secondary storage, they do provide us with what we need. In practice, however, the blocks that constitute an array may be scattered around the disk. Hence, finding the block containing the element next to the last element of a previous block may not be as simple as taking the next block on disk, so the time it takes to serve the individual memory transfers may be higher than when controlling the layout directly. It is still considered a constant, though. Algorithm 3-2 computes the sum of N elements.

Algorithm 3-2. CO_sum(Array A)

```
sum = 0;
for( int i=0; i!=N; ++i )
    sum += A[i];
```

It really could not be simpler. Notice that the assumption that $N = cB$, for some positive integer c made in the analysis of Algorithm 3-1, is not needed here. The discussion of the array construct above, gives us that Algorithm 3-2 incurs no more than $N/B+2$ memory transfers, which is asymptotically optimal. Further, the work done is $\mathcal{O}(N)$ which is also optimal, so Algorithm 3-2 is optimal in the cache-oblivious model.

The stream concept is the same as in the external memory mode, except in the cache-oblivious model, its description is simpler; we can use arrays to realize streams, either an entire array or just a part of an array, called a *subarray*. The term buffer now also simply applies to an array or a subarray, used to store elements; elements can be inserted into or extracted from buffers in a streaming fashion, incurring B^{-1} memory transfers per such operation amortized

Binary Search and the van Emde Boas Layout

For binary search in the cache-oblivious model, we use search trees, we also need to consider the layout. The standard way to do this in the RAM model is to lay out the nodes in-order, that is, simply keep the elements in an array in sorted order. This way, the rank of an element is also the position in the array, so the number by the nodes in Figure 3-2 indicates the position of the node in the array. Unlike with scanning-type problems, however, it turns out to be insufficient to simply adapt the RAM model algorithm and use an array to guarantee locality; the only place we gain from this is at the bottom of the tree. More precisely, the subtrees of height $\Omega(\log B)$ at the bottom of the tree will be stored contiguously and elements within them can all be accessed after incurring one memory transfer, however, no level above that exhibit locality. Thus, a root to leaf traversal incurs $\mathcal{O}(\log N - \log B)$ memory transfers, which is suboptimal.

It is old wisdom in computer science that dealing with datasets in a recursive fashion yields good locality. The intuition behind this is that when the problem size becomes small enough to fit in cache or in a single block, we can likely solve them optimally and combining solutions to subproblems are often trivial. What we need for binary search is a way to recurse on the paths from the root to the leaves. van Emde Boas first presented a way to recurse on trees vertically, leading to an $\mathcal{O}(\log\log U)$ query time priority queue [EKZ77]. The idea was to build a heap of elements from a universe of size U recursively from heaps of size $\sqrt{U}$, denoted bottom[0,1,..., $\sqrt{U}$-1], and use a single heap of size $\sqrt{U}$, the top, to represent the presence of elements in a given bottom-heap. A query would then first go to the top and then to one of the bottom heaps. The binary search tree equivalent of this structure is a tree that is conceptually cut in half at the middle level of edges. This is so far only conceptual; the search procedure is still the same – querying the *top tree* amounts to the first half of the root to leaf traversal and locates the *bottom tree* in which to proceed. Querying that bottom tree is the rest.

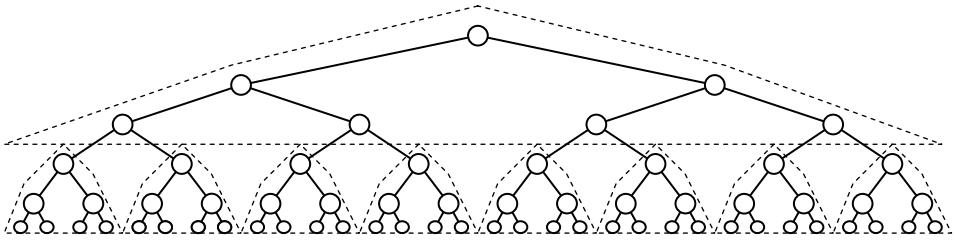


Figure 3-3. A binary search tree of size 63 split into a top-tree of size 7 and eight bottom trees also of size 7.

To apply this to get better locality, we use the concept of top and bottom trees in the storage of the tree. We lay out the nodes according to the van Emde Boas recursion: the $\sqrt{N}$ top elements are stored recursively at the head of an array and after them, in no

particular order, the elements in the $\sqrt{N}$ bottom trees are stored recursively. This particular way of laying out a tree is now called the *van Emde Boas layout*. The effect of this is that the nodes on the first half of the path from the root to the leaves are stored contiguously, as are the nodes on the second half. To realize that using the van Emde Boas layout suffices to get optimal search cost, we conceptually follow the recursion until subtrees are of size at most B and a least $\sqrt{B}$. Such a tree has height at least $\frac{1}{2}\log B$. Nodes of subtrees of the recursion are stored contiguously and so these elements can occupy at most two blocks. On the path from the root to the leaves, we thus visit no more than $2\log N/\log B$ such trees and each visit costs no more than two memory transfers for a total of $4\log_B N$ memory transfers.

3.2 External Memory Sorting

Let us now turn to the problem of sorting. In this section, we study sorting in the external memory model. Chapter 4 is dedicated to optimal sorting in the cache-oblivious model.

3.2.1 Lower Bound

An adversary argument similar to the one in the previous section proves a lower bound on the number of I/Os incurred by a correct sorting algorithm. This proof was done in [AV88], with the introduction of the external memory model, in the general case that included parallel disks. We will here show the bound in the case of a single disk ($P = 1$). For the proof, we shall need the following lemmas:

Lemma 3-1. $n \log n - \frac{n-1}{\ln 2} \leq \log(n!) \leq n \log n$.

Proof. We have

$$\log(n!) = \log \prod_{i=1}^{n} i = \sum_{i=1}^{n} \log i . \tag{3.2}$$

Since $\log$ is a continuous increasing function, the following bounds hold

$$\begin{aligned} &\int_{2}^{n+1} \log(i-1) di \leq \sum_{i=1}^{n} \log i \leq \int_{1}^{n+1} \log i di \Leftrightarrow \\ &n \log n - \frac{n-1}{\ln 2} \leq \sum_{i=1}^{n} \log i \leq (n+1)\log(n+1) - \frac{n}{\ln 2} \leq n \log n \end{aligned} \tag{3.3}$$

□

Lemma 3-2. Assuming $m \geq 2b$, $\log \binom{m}{b} \leq 3b \log m/b$.

Proof. We have

$$\log\binom{m}{b} \le \log\left(\frac{m^b}{b!}\right) = b\log m - \sum_{i=1}^{b}\log i\,. \tag{3.4}$$

Using (3.3), we get

$$\begin{aligned}\log\binom{m}{b} &\le b\log m - b\log b + \frac{b-1}{\ln 2}\\ &\le b\left(\log m - \log b + \frac{1}{\ln 2}\right)\\ &= b\left(\log {}^{m}\!/_{b} + \log e\right)\\ &\le 3b\log {}^{m}\!/_{b}\end{aligned} \tag{3.5}$$

with the last inequality using $m \ge 2b \ge b\log e$. □

Theorem 3-1. Assuming $M \ge 2B$, a correct sorting algorithm must incur $\Omega(N/B\log_{M/B}(N/B))$ I/Os in the external memory model.

Proof. As argued in the previous section, a correct sorting algorithm must be able to exclude all but the one permutation that will bring elements in order. The only operation in the RAM model that decreased the number of possible permutation was a comparison. In the external memory model, the only operations allowed are reads and writes; however, having done a read, an external memory algorithm may exclude many more than half of the permutations. When a block is read in, the position of the B elements in the block among all M elements already in memory may be determined, reducing the number of permutations by a factor $\binom{M}{B}$. Furthermore, when a block first gets read in (or when it is read in later, but no more than once), the algorithm may determine the position of all elements in the blocks among themselves, eliminating a further factor of $B!$ permutations. We call the latter reading an untouched block, while the other reads read touched blocks. The process of writing a block back to disk does not reduce the number of possible permutations.

Let $\varphi(t\text{-}1)$ denote the number of remaining possible permutations after t-1 reads or writes. Depending on the type of operation the t'th is, $\varphi(t) \ge \varphi(t\text{-}1)/X$ possible permutations remain, with

- $X = \binom{M}{B}$, in case of a read of a touched block,
- $X = \binom{M}{B}B!$, in case of a read of a untouched block, and
- $X = 1$, in case of a write.

Since we can read an untouched block no more than N/B times, we get

$$\varphi(t) \geq \frac{\varphi(0)}{\binom{M}{B}^t (B!)^{N/B}} \tag{3.6}$$

With $\varphi(0) = N!$, we are interested in the smallest t, such that $\varphi(t) \leq 1$:

$$\begin{aligned} 1 &\geq \frac{N!}{\binom{M}{B}^t (B!)^{N/B}} \Leftrightarrow \\ t \log \binom{M}{B} + N/B \log(B!) &\geq \log(N!) \end{aligned} \tag{3.7}$$

Using Lemma 3-1 and Lemma 3-2, we get

$$\begin{aligned} & t \log \binom{M}{B} + N/B \log(B!) \geq \log(N!) \Rightarrow \\ & t\left(3B \log M/B\right) + N/B\left((B+1)\log B - \frac{B}{\ln 2}\right) \geq N \log N - \frac{N-1}{\ln 2} \Rightarrow \\ & t\left(3B \log M/B\right) \geq N \log N - N\frac{B+1}{B} \log B + N \log e \Rightarrow \\ & t \geq \frac{N \log N/B + \log e}{3B \log M/B} \end{aligned} \tag{3.8}$$

as desired. □

3.2.2 Multiway Merge Sort

A variant of merge sort achieves optimal complexity in the external memory model. In this thesis, we refer to it as multiway mergesort. It relies on an abstract data structure we call a *k-merger*. A k-merger is capable of merging up to k sorted streams; input streams are attached to the merger, and when invoked, the merger outputs the elements of the input streams in one sorted stream. An efficient implementation would use an efficient priority queue, such as a binary heap, of size k, storing pairs (s,e) of stream identifiers s and elements e with e being a copy of the next element of the stream identified by s. The priority of the pair is e. Additionally, from each stream a block of elements is kept in memory. An element is merged by doing a delete_min on the queue, returning (s,e), extract the next element e' from s, insert (s,e') into the queue and output e. According to the discussion of Algorithm 3-1, all accesses are in a streaming fashion, provided memory has the capacity to hold k blocks and the priority queue.

Multiway mergesort works in two phases. First, the "run formation" phase reads in the input, one memory load at a time, sorts the memory loads internally, and outputs the sorted run. Second, M/B sorted runs are merged using an M/B-merger, reducing the

number of sorted runs by a factor of M/B. More accurately, an $(M/(B+1)\text{-}1)$-merger should be used to make room for the queue and leave one block for streaming the output. This is repeated until one sorted run remains. The procedure is illustrated in Algorithm 3-3. For simplicity, we assume $N = c_1 M = c_1 c_2 B$ for some positive integers c_1 and c_2.

Algorithm 3-3. multiway_mergesort

```
runs = N/M
repeat runs times do
    read in M/B blocks
    sort the M elements
    write out M/B blocks
od
while runs > 1 do
    repeat for groups of M/B runs do
        construct an M/B-merger
        attach M/B runs as its input streams
        merge all elements in the runs
    od
    runs = runs/(M/B)
od
```

That Algorithm 3-3 is indeed optimal, is given by

Theorem 3-2. Algorithm 3-3 incurs $\mathcal{O}(N/B \log_{M/B}(N/M))$ I/Os in the external memory model.

Proof. The first phase incurs N/B reads and N/B writes in total. So does each iteration in the second phase. After the first phase, there are N/M runs. An iteration in second phase reduces the number of runs by a factor of M/B, thus there are $\log_{M/B}(N/M)$ iterations in the second phase for a total of

$$\begin{aligned}\frac{2N}{B} + \frac{2N}{B}\log_{M/B} N/M &= \frac{2N}{B}\left(1 + \log_{M/B} N/M\right) \\ &= \frac{2N}{B}\left(\log_{M/B} M/B + \log_{M/B} N/M\right) \\ &= \frac{2N}{B}\log_{M/B} N/B\end{aligned} \tag{3.9}$$

I/Os, which is correct in asymptopia, even if $N = c_1 M = c_1 c_2 B$ does not hold. □

Note that, only when $N \gg M$ does the logarithmic term become significant. It might be interesting to see what the complexity is for more sensible N. When $N \leq M$, the second phase is never invoked, so a total of than $2\lceil N/B \rceil$ I/Os are incurred. For all $N \leq M(M/(B+1)\text{-}1)$, only one iteration is needed in the second phase, and we incur $4\lceil N/B \rceil$ I/Os. With M elements taking up one half gigabyte and $B+1$ elements 4kB, the second condition is met for all input that takes up less than 64 TB. Assuming one millisecond per I/O, sorting a data set of that size would take at least two years.

3.2.3 Distribution Sort

Distribution sorting is a recursive process, like mergesort, but it solves a different problem at each level [Knu98]. A set of S-1 partitioning elements are used to partition the input in S disjoint *buckets*. All the elements in one bucket are smaller than elements in the next bucket.

The adaptation of distribution sort to the external memory model follow that of mergesort; problems larger than M are split into problems a factor of M/B smaller. Distribution generates subproblems a factor of S smaller, so we choose $S = \Theta(M/B)$. When problems become smaller than M, we incur no more I/Os, so we get $\mathcal{O}(\log_{M/B} N/B)$ levels of recursion. If we can distribute N elements evenly into M/B buckets using $\mathcal{O}(N/B)$ I/Os, distribution sort becomes optimal in the external memory model. [AV88] shows how to distribute N elements into $\sqrt{M/B}$ buckets (the square root effectively only doubling the number of recursion levels), however the constants involved in the $\mathcal{O}(N/B)$ bound are much larger than those of multiway merge sort. It involves a pre-sorting phase, that much like multiway mergesort sorts memory loads (albeit in memory, thus incurring only $2N/B$ I/Os) *before* the elements are distributed in buckets and then sorted recursively (again), so the instruction count also has a high leading term constant.

3.2.4 Cache-Oblivious Sorting

The same lower bounds hold for a cache-oblivious algorithm as for an external memory algorithm, so the goal of optimal cache-oblivious sorting algorithms is to match the I/O bound of Theorem 3-1 and the work bound of $\mathcal{O}(N\log N)$, while not referring explicitly to the memory system as Algorithm 3-3 does. Using the tall cache assumption that $M = \Omega(B^{(d+1)/(d-1)})$, we can rephrase this bound in the cache-oblivious model. We have

$$\begin{aligned}
\Omega\left(\frac{N}{B}\log_{M/B} N/B\right) &= \Omega\left(\frac{N}{B}\frac{\log N/B}{\log M/B}\right) \\
&= \Omega\left(\frac{N}{B}\frac{\log N/B}{\log M^{1-\frac{d-1}{d+1}}}\right) \\
&= \Omega\left(\frac{d+1}{2}\frac{N}{B}\log_M N/B\right) \\
&= \Omega\left(d\frac{N}{B}\left(\log_M N - \log_M B\right)\right) \\
&= \Omega\left(d\frac{N}{B}\log_M N\right)
\end{aligned} \tag{3.10}$$

Since all RAM algorithms are also cache-oblivious algorithms, the popular and efficient quicksort is also a cache-oblivious sorting algorithm. Indeed, it follows recursive divide-and-conquer strategy, which is intuitively good for cache locality. Efficient implementations use a constant time median approximating scheme to partition the elements in a set of small elements and a set of large elements, and then

recurse on each set. The partitioning of n elements can be done with $\mathcal{O}(n)$ operations and $\mathcal{O}(n/B)$ memory transfers without knowing B. At each level, a total of N elements are partitioned for a total of $\mathcal{O}(N)$ operations and $\mathcal{O}(N/B)$ memory transfers. When considering the work done by the algorithm, the recursion continues until a constant number of elements are left in the set, for a total of $\mathcal{O}(\log N)$ recursion levels. However, for the I/O complexity, by virtue of the recursive structure, when $n < M$ the partitioning will incur no more memory transfers for that particular subproblem. The number of recursion levels to consider for the I/O complexity is thus expected to be the number of times N can be halved before it becomes smaller than M, namely $\mathcal{O}(\log N/M)$. Hence, quicksort does $\mathcal{O}(N \log N)$ work, incurs $\mathcal{O}(N/B \log(N/M))$ memory transfers and is thus fortunately not asymptotically optimal. However, it does come very close, which it is a testament to the efficiency of quicksort.

We saw that in practice, multiway mergesort performs no more than $4\lceil N/B \rceil$ I/Os. [LL99] estimates the factor on the $\lceil N/B \rceil$ term in quicksort to be $2\ln(N/M)$ on uniform data when counting cache-misses. With M half a gigabyte and N two gigabytes and including writes, this constant is roughly 5.54, so when sorting less than two gigabytes, quicksort incurs no more than 38.6% more memory transfers than the optimal multiway mergesort. Obviously, on the lower levels of the memory hierarchy $2\ln(N/M)$ can get higher for reasonable N, however the penalty of being suboptimal on those levels are not as high.

Chapter 4

Cache-Oblivious Sorting Algorithms

As Frigo *et al*. presented the novel model of cache-oblivious computing they also presented a host of optimal cache-oblivious algorithms and data structures [FLPR99]. Aside from Fast Fourier Transformation, Matrix Multiplication, and Matrix Transposition they presented two optimal sorting algorithms. Both are in essence cache-oblivious variants of the two sorting algorithms presented in the previous chapter. What follow is a thorough presentation of cache-oblivious merge sort, dubbed funnelsort.

4.1 Funnelsort

The cache-awareness of multiway merge sort is twofold; initial runs are of size M, and in merges runs with an M/B-merger.

In multiway mergesort, we could simply implement the merger using a binary heap, or an equivalent, as a priority queue; but without knowing M or B, we do not know what size it should be, nor do we know if it can even fit in cache. If a binary heap, for instance, cannot fit in cache, its I/O performance decreases drastically. Hence, we need a merger structure that, no matter what M is, can merge efficiently. The key to this is, as with binary searching, recursion, albeit in a slightly different form. However, simply adapting the layout of a binary heap turns out to be insufficient; indeed constructing an I/O optimal heap is non-trivial. Fortunately, we do not need to go so far.

4.1.1 Merging with Funnels

The cache-oblivious equivalent of a multiway merger is a *funnel*. A k-funnel is a static data structure capable of merging k sorted input streams. It does so by merging k^d elements at a time, for some $d > 1$. It is arranged as a tree, with two notable features: it is stored recursively according to the van Emde Boas layout, with the top and bottom trees them selves being (sub-) funnels; secondly, along each edge, it keeps a buffer. Intuitively, we need such buffers to amortize the cost of using a (sub-) funnel, in case it is so big it cannot be operated with a constant number of memory transfers.

Despite it being a rather young data structure, the funnel has already undergone a couple of changes. After its introduction in 1999 along with funnelsort, Brodal and Fagerberg [BF02a] presented a conceptually simpler version, dubbed lazy funnelsort, and in this thesis, we will introduce further simplifications. It seems natural to describe the original funnel and then introduce the modifications, to establish an intuition of why the structure is optimal. The analysis will follow the final modifications.

The Eager Funnel

The original k-funnel[1] merged k^3 elements at a time. It consists of $\sqrt{k}$ $\sqrt{k}$-funnels L_0, L_1, …, $L_{\sqrt{k}-1}$, at the bottom, each connected to a $\sqrt{k}$-funnel R, at the top, with edges, that contain buffers, B_0, B_1, … $B_{\sqrt{k}-1}$. We will discuss the case of non-square orders below. The buffers doubles as both the output of the bottom funnels as well as input for the top funnel. Figure 4-1, shows a 16-funnel consisting of five 4-funnels and four buffers. At the base of the recursion, at the nodes of the tree, we have constant sized binary or ternary mergers. Note, that a k-funnel in it self does not have input or output buffers, though it may sometimes be convenient to conceptually include e.g. an output buffer in a funnel. The funnel is stored at contiguous locations in memory, according the van Emde Boas layout: first, the top tree is laid out recursively, immediately thereafter comes B_0 followed by a recursive layout of L_0 followed by B_1 and so on.

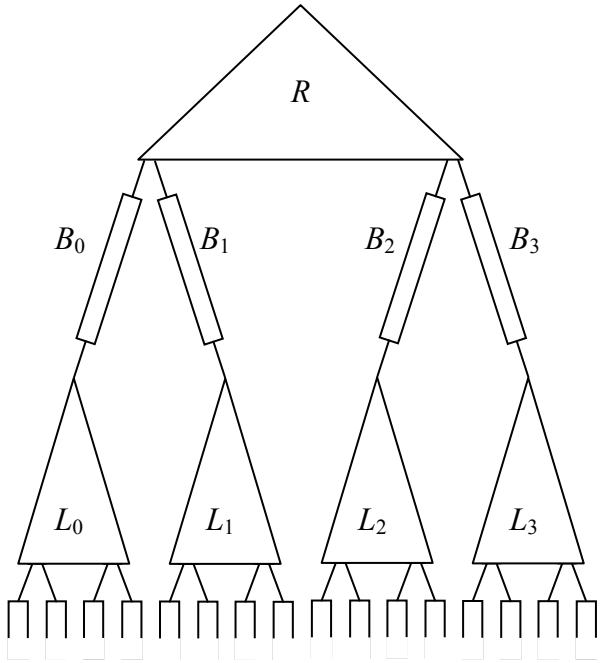


Figure 4-1. A 16-funnel with its 16 input streams.

To operate a funnel, we use a recursive procedure invoke, illustrated in Algorithm 4-1. It reads a total of k^3 elements form the funnel's sorted input streams, and output k^3 elements in sorted order. For the amortization argument to work, there must initially be at least k^2 elements in each input buffer, but as the merging progresses, there will be fewer and fewer elements in the input. If at some point a (sub-) funnel cannot fulfill the obligation to output k^3 elements, due to insufficient input, it outputs what it can, is marked exhausted, and will not be invoked again. This is done to avoid futile descends into bottom funnels, that may not produce enough elements. There will be at most one invocation, per funnel, that outputs less k^3 than elements, so it will not influence the asymptotic analysis.

[1] ... was originally called a k-merger, but since a merger, at least in my mind, is a broader class of data structures, I will use the term k-funnel for this kind of merger.

Algorithm 4-1. invoke(k-Funnel F)

```
if k is 2 or 3 then
    merge "manually"
else
    repeat k^3/2 times do
        for each buffer B_i, 0 ≤ i < k do
            if B_i less than half full and L_i is not exhausted then
                invoke(L_i)
        invoke(R)
    od
fi
if less than k^3 elements was output then
    mark F exhausted
```

The recursion of invoke follows the layout; when a funnel is invoked, it invokes the top funnel $k^{3/2}$ times, each time contributing $k^{3/2}$ elements to the output, for a total of k^3 elements. Before each invocation, however, it checks whether there are enough elements in the buffers (the input of the top funnel) and for each buffer with less than $k^{3/2}$ elements, it invokes the bottom funnel with that particular buffer as its output. For this to work, the buffers are implemented as FIFO queues and have a capacity of at least $2k^{3/2}$. In Figure 4-1, each of the four buffers has a capacity of 128 elements. This will insure that if there are at most $k^{3/2}$ elements there is room for an additional $k^{3/2}$, while at the same time guarantee that there will be enough elements to output $k^{3/2}$ elements, even in the extreme case the elements to be output all come from the same input stream.

It can be proven (see Section 4.1.2 below) that a k-funnel takes up $\mathcal{O}(k^3)$ contiguous memory locations. So, intuitively, the reason the funnel works is that we may have to "pay" the memory transfers to get it going (about $\mathcal{O}(k^2/B)$), but we get $\Omega(k^3)$ elements out of it (paying a total price of $\Omega(k^3/B)$). Additionally, if the funnel is too big to fit in cache, regardless of its size, there will be some level of the van Emde Boas recursion, at which the subfunnel will fit in cache. For that level, the previous argument applies, and as it turns out, there are not too many levels.

The above description may seem as a simple way of merging; however, there a couple of inherent practical problems. Most severe is perhaps the subtle dependencies between the buffer sizes and the order of both the top and bottom funnels. Given that we in practice cannot split a k-funnel into $\sqrt{k}$ -funnels, simply because k may not be square, we run into rounding issues: The standard way of avoiding rounding problems would be to choose k as the smallest order that ensures only square funnels throughout the funnel. This order however is one that gives the tree a power-of-two height, that is, it needs to be of order power-of-power-of-two, which would make the funnel too big in the worst case. So a k-funnel is in general comprised of j $\left\lceil\sqrt{k}\right\rceil$-funnels and $\left(\left\lceil\sqrt{k}\right\rceil - j\right)\left\lfloor\sqrt{k}\right\rfloor$-funnels (perhaps one less) aside from the top funnel, for some $j \leq \left\lceil\sqrt{k}\right\rceil$. Now the buffers may be too small either to hold the output of a bottom funnels or to ensure enough elements for the top funnel. A solution to this problem is, after the order of the top and bottom funnels have been determined, the i'th buffer should be of size $2m_i^{3/2}$, where m_i is at the maximum of the order of the i'th bottom funnel and the top funnel. But now a single invoke of a bottom funnel will not insure, that the buffer is at least half filled, so the algorithm need to sometimes do more than a single invoke.

In addition, the apparent simplicity of the merging phase may be deceiving; we have explicitly made certain prior to the invocation of any funnel, that there will be enough elements in the input to produce the output and may merge k^3 elements, without checking the state of the buffers. This is false. The issue of exhaustion cannot be controlled, because it depends on the input, so any subfunnel may at any time become exhausted and its output buffer only be partially filled. The funnel that uses this buffer as input has to consider this, in effect making it paranoid, always checking to see if there indeed are more elements.

Aside from these concerns, there is the practical issue of doing the actual layout. While doing a van Emde Boas layout of a binary tree with fixed-sized and -typed nodes may be trivial, doing it with variable sized buffers and mixed data types is not. Suffice it to say, that on some hardware architectures, certain data types cannot be placed at arbitrary memory locations. This is known as alignment. For example, a double precision floating-point number may only be addressed, if it is placed at addresses divisible by eight, while an integer can be at any address divisible by four. If we were to merge doubles with a funnel, we would have to lay out buffers capable of containing doubles intermixed with nodes containing pointers and Booleans. Say a node consists of six pointers of four bytes each and a byte for the exhausted flag, for a total of 27 bytes. Now we cannot simply place a buffer, a node, and then another buffer contiguously. Simply using four bytes for the exhausted flag does not help. To make matters worse, to our knowledge, no language supports a simple way of determining alignment requirements of data types.

The Lazy Funnel

Many of these issues were eliminated with the introduction of the *lazy funnel* [BF02a]. The modification lies primarily in the operation of the funnel. The lazy funnel generates the output of bottom funnels lazily, in the sense that they are invoked as needed, when there are no more elements in the buffers, as opposed to explicitly checking the state of the buffers and then perhaps invoking a bottom funnel. This means, in turn, that there is no longer operationally a need to follow the layout in the recursion, and thus no need for a notion of top and bottom funnels; all base funnels (the binary and ternary mergers) are invoked the same way, through a procedure called `lazy_fill`, illustrated, for a binary node, in Algorithm 4-2. Note that if one of the children is exhausted, the corresponding input may not contain any elements so the head of that input may be undefined and should be ignored or taken to be the largest possible element.

Algorithm 4-2. lazy_fill(Node v)

```
while v's output buffer is not full do
    if left input buffer empty and left child of v not exhausted then
        lazy_fill(left child of v)
    if right input buffer empty and right child of v not exhausted then
        lazy_fill(right child of v)
    if head of left input < head of right input then
        move head of left input to output
    else
        move head of right input to output
od
if left input buffer empty and left child of v exhausted and
        right input buffer empty and right child of v exhausted then
    mark as v exhausted
```

This allows for two structural simplifications: the lazy funnel is now simply a balanced binary tree, no mix of binary and ternary mergers. Secondly, there are no subtle dependencies between the size of the buffers and the funnels using them as input or output. The storing of the funnel is still done according to the van Emde Boas layout as are the capacity of the buffers still a function of the order of the subfunnel that has it as its output. The shape of the function, however, now only has an analytical significance, so we are free to choose from a larger class of functions. The analysis was originally carried out with k-funnels generating output of size k^d, with $d \geq 2$.

These simplifications have since paved the way for the use of the funnel data structure as a key element in many cache-oblivious algorithms and data structures, such as the distribution sweeping class of geometrical algorithms [BF02a] and the funnel heap [BF02b]. The many uses of the funnel underline the need for good solid implementation.

Two-Phase Funnel

The practical issues that remain in the lazy funnel are the special attention to the exhaustion flag and the complex layout. The *two-phase funnel* resolves these issues. As for the latter, it is simply not needed; the two-phase funnel does not care about the layout. As with the lazy funnel, the van Emde Boas recursion is still used for determining the capacity of the buffers, so the layout of the original funnel structure is not completely gone. With the two-phase funnel, the use of controlled layout is optional. Whether it influences performance in practice will be investigated in Chapter 6.

As for the former, we first look at scheduling of the nodes. It would seem that the merging proceeds until either the output is full or *exactly one* of the node's inputs is empty, in which case lazy_fill is called on the corresponding child. There are exactly two cases, where this is not true. One is in the initial funnel, where all buffers are empty. Invoking lazy_fill on the root will not merge until one input is empty, simply because both are empty. The other case is when the last time lazy_fill was called on it, the node below an input, was exhausted and could thus not produce any elements. This is important, since it gives an indication of when a merger may be exhausted; we will thus maintain the invariant that *as we start filling, buffers that are empty, are the output of exhausted mergers, and should thus be ignored*. This is not true, however, in the initial funnel, but that can be fixed by introducing a special warm-up phase. This phase is the

part of the merging that takes place before the first call to lazy_fill starts outputting elements. It consists essentially of head-recursive calls to lazy_fill, as Algorithm 4-3 illustrates.

Algorithm 4-3. warmup(Node v)

```
if v not leaf then
    warmup(left child of v)
    warmup(right child of v)
fi
fill(v)
```

If, in case all input streams are empty, the invocation of fill simply does nothing and returns, the invariant holds; warmup guarantees that fill is called on nodes, only after fill has been called on all the children. Leaf nodes are exhausted if and only if no elements are in the input. If so, fill will do nothing and return, thus leaving an empty input buffer for the parent. This in turn will be taken to mean that the leaf is exhausted. Inductively, if all children of an internal merger have returned with empty buffers, they are all exhausted, and since no elements are in the buffers, the internal merger is exhausted. The same argument applies in the second phase, when elements are actually output from the root of the funnel. Note that the scheduling of nodes are the same as in the lazy funnel, thus this is a mere algorithmic simplification and not an operational one. The modified version of lazy_fill is illustrated in Algorithm 4-4. Note, also, the potential performance gained through the stronger invariant. While Algorithm 4-2 evaluates four conditional branches per element merged, Algorithm 4-4 only evaluates three. In addition, it explicitly needs to check the exhaustion flag.

Algorithm 4-4. fill(Node v)

```
if both inputs are non-empty then
    while v's output buffer is not full do
        if head of left input < head of right input then
            move head of left input to output
            if left input buffer empty then
                fill(left child of v)
        else
            move head of right input to output
            if right input buffer empty then
                fill(right child of v)
        fi
    od
else if exactly one input buffer empty then
    move as many elements as possible from the other input to the output
    if input buffer got empty then
        fill(corresponding child of v)
else
    return
fi
```

Generalization of base nodes from binary to multiway is now also straightforward; the invariant holds as long as fill simply returns, if no stream contain elements and it only considers non-empty streams for merging. Using z-way base mergers corresponds

to stopping the van Emde Boas recursion before we reach trees of height one; stopping at e.g. height two would yield a funnel with four-way base mergers. In the analysis that follows, we will consider the use of z-way base mergers, as well as a slightly larger class of buffer size functions, namely αk^d with α and d being constants and k the order of the funnel below the buffer.

4.1.2 Funnel Analysis

In this section, we analyze the complexity in the cache-oblivious model of filling the output buffer of a funnel, using a two-phase funnel. First, the total size of the funnel needs to be bounded; this is important for determining when a funnel fits in cache. For this, we follow the van Emde Boas recursion. Recall that the van Emde Boas recursion is actually a horizontal split of the tree at depth half the height h, the order of a funnel being z^h, so the size of the output becomes αz^{hd}.

Lemma 4-1. Assuming $d \geq 2$ and $k = z^h$ for some $h > 0$, a k-funnel spans at most $S(k) \leq \gamma k^{\frac{d+1}{2}} B^{-1} + 4k$ blocks, with $\gamma = 2.00033 \frac{\alpha z^{d+1}}{1 - z^{-2}}$.

Proof. First, consider the number of blocks needed to hold the buffers. A buffer is an array of $\beta(h) = \alpha z^{hd}$ elements and occupies as such no more than $\beta(h)/B+2$ blocks, the extra two being in the case of the ends reaching into parts of other blocks. For now, we assume that buffers take up $\beta(h)/B$ blocks, and will compensate later. A funnel of height h has by convention of this thesis a top tree of height $\lfloor h/2 \rfloor$[1]. Aside from the buffers in the top and bottom funnels, such a funnel has at most $N(h/2) = z^{h/2}$ buffers. Each of these has a capacity determined by the bottom funnel, namely $\beta(\lceil h/2 \rceil) \leq \beta((h+1)/2)$, since the bottom funnel has height $\lceil h/2 \rceil$. Correspondingly, there are at most $N((3h+1)/4)$ buffers in the bottom funnel and $N(h/4)$ in the top funnel. The recursion continues in this way, down to funnels of height one, so the total number of blocks used by buffers of a funnel of height h is

$$\begin{aligned} s(h) &\leq N(h/2)\,\beta\left(\tfrac{h+1}{2}\right)B^{-1} + \left(N\left(\tfrac{3h+1}{4}\right) + N(h/4)\right)\beta\left(\tfrac{h+3}{4}\right)B^{-1} \\ &\quad + \left(N\left(\tfrac{7h+5}{8}\right) + N\left(\tfrac{5h+3}{8}\right) + N\left(\tfrac{3h+1}{8}\right) + N(h/8)\right)\beta\left(\tfrac{h+7}{8}\right)B^{-1} + \cdots \qquad (4.1) \\ &\leq \sum_{i=1}^{\log h}\left[\beta\left(2^{-i}h+1\right)B^{-1}\sum_{j=1}^{2^{i-1}}\left(N\left(2^{-i}h(2j-1)+2j\right)\right)\right] \end{aligned}$$

Using the bound

$$\sum_{j=0}^{n-1} x_i^{\,j} = \frac{x_i^{\,n}-1}{x_i-1} = \frac{x_i^{\,n}-1}{x_i\left(1-1/x_i\right)} \leq \frac{1}{1-\max_i\left(x_i\right)^{-1}}\, x_i^{\,n-1}, \qquad (4.2)$$

we get the bound of the inner sum

[1] [BDFC00] uses the convention that search trees of height h have bottom trees of height $\lceil h/2 \rceil$. However, this rounding could lead to very large buffers just below the root, when the tree has height 2^j+1 for some integer j.

$$\begin{aligned}\sum_{j=1}^{2^{i-1}} N\left(2^{-i}h(2j-1)+1\right) &= \sum_{j=1}^{2^{i-1}} z^{2^{-i}(h(2j-1)+2j)} \\ &= z^{-2^{-i}h}\sum_{j=1}^{2^{i-1}} z^{2^{1-i}(h+1)j} \\ &\le z^{-2^{-i}h}\left(\frac{1}{1-z^{-2}}z^{2^{1-i}(h+1)2^{i-1}}\right) \\ &\le \frac{z}{1-z^{-2}}z^{\left(1-2^{-i}\right)h}\end{aligned} \tag{4.3}$$

and the outer sum becomes

$$\begin{aligned}s(h) &\le \sum_{i=1}^{\log h}\frac{z}{1-z^{-2}}z^{\left(1-2^{-i}\right)h}\beta\left(2^{-i}h+1\right)B^{-1} \\ &= \frac{z}{1-z^{-2}}\sum_{i=1}^{\log h} z^{\left(1-2^{-i}\right)h}\left(\alpha z^{\left(2^{-i}h+1\right)d}B^{-1}\right) \\ &= \frac{z}{1-z^{-2}}\frac{\alpha}{B}\sum_{i=1}^{\log h} z^{\left(1-2^{-i}\right)h+\left(2^{-i}h+1\right)d} \\ &= \frac{z}{1-z^{-2}}\frac{\alpha z^d}{B}\left(z^{h\frac{d-1}{2}}+\sum_{i=2}^{\log h} z^{h\left(2^{-i}(d-1)+1\right)}\right) \\ &\le \frac{z}{1-z^{-2}}\frac{\alpha z^d}{B}\left(z^{h\frac{d+1}{2}}+z^{h\frac{d+3}{4}}\log h\right) \\ &= \frac{z}{1-z^{-2}}\frac{\alpha z^d}{B}\left(1+z^{h\frac{1-d}{4}}\log h\right)z^{h\frac{d+1}{2}}\end{aligned} \tag{4.4}$$

With $d,z \ge 2$, the following bound has been estimated numerically:

$$z^{h\frac{1-d}{4}}\log h < 1.00033\,.$$

The term is actually bounded by a finite constant, for all $d > 1$. Realistically, as the d approaches 1, the term becomes bounded by $\log h$, which for $\alpha \ge 1$, $z \ge 2$ and the size of the output $\alpha z^{hd} < 2^{64}$, is bounded by 6.

Each buffer may extend onto two additional blocks and each node may span at most two blocks (assuming a block is big enough to hold at least one). Since there are no more than z^h buffers or nodes, nodes and buffer-ends can contribute with no more than $4z^h$ blocks, which completes the proof. □

Note, that for constant α and z, $S(k)$ is $\mathcal{O}(k^{(d+1)/2}/B+4k)$. The fact, that there is no need to layout the funnel in contiguous memory locations not only makes life easer on the implementer, it also provides for greater flexibility which is important when using the funnel in dynamic data structures, such as the funnel heap [BF02b].

Now we are ready to prove the I/O bound of fill.

Theorem 4-1. Assuming $M \geq (cB)^{(d+1)/(d-1)}$, for some $c > 0$, and the input streams contain a total of αk^d elements a k-funnel performs $\mathcal{O}(k^d \log_M(k^d)/B + k)$ memory transfers during an invocation of fill on the root, for fixed α and z.

Proof. For this proof, we also conceptually follow the van Emde Boas recursion, but this time only until we reach a subfunnel of order j, such that $\gamma j^{(d+1)/2} \leq \varepsilon M$, with ε being the solution to

$$\left(\frac{\varepsilon}{\gamma}\right)^{\frac{2}{d+1}} = \frac{c}{5}(1-\varepsilon). \tag{4.5}$$

For finite $c > 0$ and $d > 1$ we have $0 < \varepsilon < 1$. Figure 4-2 shows the recursion in case the k-funnel is too big to fit in cache.

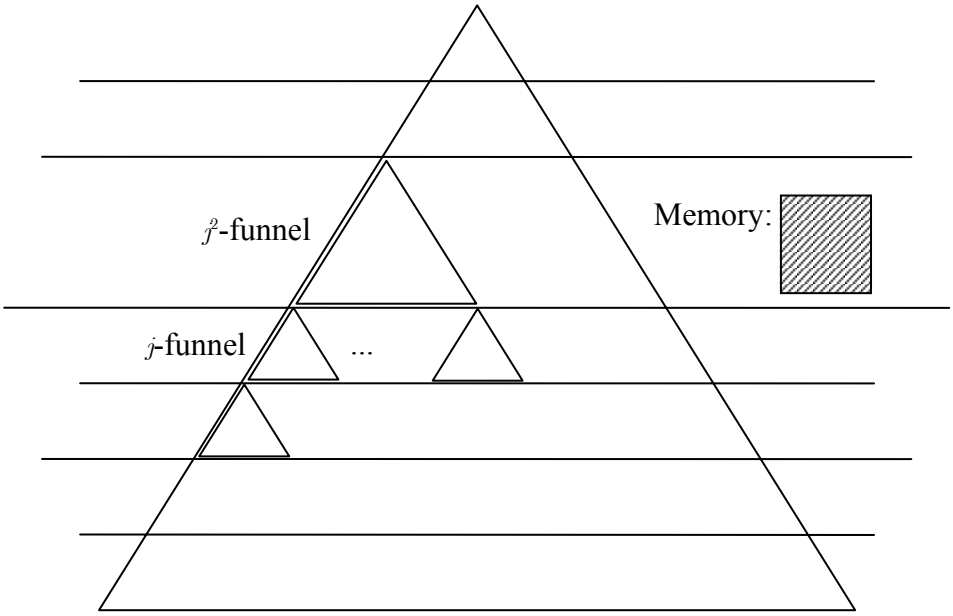


Figure 4-2. A k-funnel at the level of detail, where it consists of j-funnels. The shaded box indicates the relative size of memory.

A j-funnel has two important properties. First, the blocks spanned by a j-funnel, as well as a block from each of its input streams, all fit in cache. Second, if the entire k-funnel does not fit in cache, a j-funnel outputs at least a (positive) factor of B more elements, than there are input streams. The first property states that $S(j)+j \leq M/B$, which is proved using definition of j

$$\gamma j^{\frac{d+1}{2}} \leq \varepsilon M \Leftrightarrow 5j \leq 5\left(\frac{\varepsilon}{\gamma}\right)^{\frac{2}{d+1}} M^{\frac{2}{d+1}} \tag{4.6}$$

and Lemma 4-1:

$$\begin{aligned} S(j) + j &= \gamma j^{\frac{d+1}{2}} B^{-1} + 5j \\ &\leq \varepsilon M/B + 5\left(\frac{\varepsilon}{\gamma}\right)^{\frac{2}{d+1}} M^{\frac{2}{d+1}} \\ &= \varepsilon M/B + 5\left(\frac{\varepsilon}{\gamma}\right)^{\frac{2}{d+1}} M \cdot M^{\frac{1-d}{1+d}} \\ &\leq \varepsilon M/B + \frac{5}{c}\left(\frac{\varepsilon}{\gamma}\right)^{\frac{2}{d+1}} M/B \\ &\leq M/B \end{aligned} \tag{4.7}$$

The second property states that $\alpha j^d > ajB$, for some a. Essentially, this is a guarantee that the funnel will be able to pay the price of touching the input streams with the elements output. By definition of j, the subfunnel on the previous recursion level could not fit in cache. That funnel is of order at most zj^2 (the factor z being in case the j-funnel is the top funnel of a funnel of odd height) so we have

$$\begin{aligned} &\gamma\left(zj^2\right)^{\frac{d+1}{2}} > \varepsilon M \Leftrightarrow \\ &\gamma z^{\frac{d+1}{2}} j^{d-1} > \varepsilon^{\frac{d-1}{d+1}} M^{\frac{d-1}{d+1}} \Leftrightarrow \\ &z^{\frac{d+1}{2}} j^{d-1} > \frac{c}{\gamma} \varepsilon^{\frac{d-1}{d+1}} B \Leftrightarrow \\ &\alpha j^d > \frac{\alpha c}{\gamma} z^{-\frac{d+1}{2}} \varepsilon^{\frac{d-1}{d+1}} jB \end{aligned} \tag{4.8}$$

Consider now an invocation of fill on the root of a j-funnel. Assuming enough elements in the input, this invocation will output at least αj^d elements. First, however, we must load the j-funnel and a block from each stream. This will cost at most $\gamma j^{(d+1)/2}/B + 5j$ memory transfers. Now the funnel can stay in cache and output the αj^d elements in a streaming fashion. When a block from an input stream has been used, the optimal replacement will read the next part of the stream read into that block; hence, the input will also be read in a streaming fashion. By the second property, the number of memory transfers per element output becomes at most

$$\begin{aligned} \frac{\gamma j^{\frac{d+1}{2}} B^{-1} + 5j}{\alpha j^d} + 2B^{-1} &\leq \left(\frac{\alpha}{\gamma j^{\frac{d-1}{2}}} + \frac{5\gamma}{\alpha c} z^{\frac{d+1}{2}} \varepsilon^{\frac{d-1}{d+1}} + 2\right) B^{-1} \\ &\leq \left(\frac{\alpha}{\gamma} + \frac{5\gamma}{\alpha c} z^{\frac{d+1}{2}} \varepsilon^{\frac{d-1}{d+1}} + 2\right) B^{-1} \end{aligned} \tag{4.9}$$

While there are a total of αk^d elements in the input of the k-funnel, and it therefore will not run empty during the merge, the input of all but the bottom-most j-funnels may run empty while it is in cache and actively merging. This may evict the funnel from memory, cause the j-funnel below the empty buffer to be loaded, and in turn reload the first funnel. However, at least αj^d elements have been merged into the buffer, and by the argument above, each of these elements may be charged the $(5\gamma/(\alpha c)z^{(d+1)/2}\varepsilon^{(d-1)/(d+1)}+\alpha/\gamma+2)B^{-1}$ memory transfers to pay what it costs to reload the funnel.

In total each element is charged $2(5\gamma/(\alpha c)z^{(d+1)/2}\varepsilon^{(d-1)/(d+1)}+\alpha/\gamma+2)B^{-1}$ memory transfers, per j-funnel it passes through. There are no more than $1+\log_j k$ j-funnels on the path from the input of the k-funnel to the root and since $\gamma z^{(d+1)/2} j^{d+1} > \varepsilon M$ implies $j > (M\varepsilon/\gamma)^{1/(d+1)}/z^{1/2}$, that number is bounded by

$$\begin{aligned}
1+\log_j k &= 1+\frac{\log k}{\log j} \\
&< 1+\frac{\log k}{\log\left(\frac{\varepsilon^{\frac{1}{d+1}}}{\gamma} z^{-1/2} M^{\frac{1}{d+1}}\right)} \\
&= 1+\frac{(d+1)\log k}{\log\left(\frac{\varepsilon}{\gamma z^{\frac{d+1}{2}}} M\right)} \\
&= 1+\log_{\frac{\varepsilon M}{\gamma z^{\frac{d+1}{2}}}} k^{d+1}
\end{aligned} \tag{4.10}$$

The total number of memory transfers caused by all elements going all the way through the k-funnel is then bounded by

$$\frac{\alpha k^d}{B} + u\,\frac{\alpha k^d}{B}\log_{\frac{\varepsilon M}{\gamma z^{\frac{d+1}{2}}}} k^{d+1} \tag{4.11}$$

with

$$u = 2\left(\frac{\alpha}{\gamma} + \frac{5\gamma}{\alpha c} z^{\frac{d+1}{2}} \varepsilon^{\frac{d-1}{d+1}} + 2\right). \tag{4.12}$$

When finally a j-funnel becomes exhausted, it may not have output αj^d elements and the elements them selves will not be able to pay for the memory transfers. However, the j-funnel is permanently marked exhausted and will thus not be invoked again. So we will only come up short once per j-funnel, and can thus charge the missing payment to the output buffer it self. Charging each position in the buffer $(5\gamma/(\alpha c)z^{(d+1)/2}\varepsilon^{(d-1)/(d+1)}+\alpha/\gamma+2)B^{-1}$ memory transfers, will account for might be missing, when the funnel below it became exhausted. However, since there are at most $\gamma k^{(d+1)/2}$ buffer positions in total, this is a low-order term. □

As for the work complexity of merging elements with a funnel, the following theorem holds:

Theorem 4-2. A k-funnel performs $\mathcal{O}(\alpha k^d z \log_z k)$ operations during an invocation of fill on the root.

Proof. During a fill on a given node, elements are moved from one level in the merger to a higher level. At each step in fill, the smallest element among the heads of the z buffers must be found and moved to the output. This costs at most z-1 comparisons and one move and the result is that the element is at a higher level. The merger has height $\mathcal{O}(\log_z k)$, so the total number of moves and comparisons are $\mathcal{O}(z \log_z k)$ per element merged. Using binary heaps to merge the z inputs of a node, we get the optimal bound of $\mathcal{O}(\log k)$.

Since we move from level to level and visits each node several times during an invocation of fill on the root of a k-funnel, we should also consider the number of *tree operations*, that is, the number of times we move from one node to another. For that, we see that moving from a node to its parent implies having filled the buffer on the edge connecting them or that the node is the root of a merger has become exhausted. In the first case, moving from a node to its parent, crossing the middle of a k'-funnel, implies having merged $\Omega\left(\beta\left(\sqrt{k'}\right)\right) = \Omega\left(\alpha k'^{d/2}\right)$ elements. We need to move a total of αk^d element past the middle, so we cross it $\mathcal{O}((\alpha k^d)/(\alpha k'^{d/2})) = \mathcal{O}(k^d/k'^{d/2})$ times. The total number of times we go from a node to its parent is thus bounded by the recurrence

$$T\left(k'\right) = \begin{cases} \mathcal{O}\left(\frac{k^d}{z^d}\right), k' \leq z \\ \left(\sqrt{k'}+1\right) T\left(\sqrt{k'}\right) + \mathcal{O}\left(\frac{k^d}{k'^{d/2}}\right), otherwise \end{cases} \tag{4.13}$$

which is dominated by the base case. There are at most $\mathcal{O}(½\log_z k)$ levels with $k' \leq z$ and since we begin and end at the root, we go from parent to child as many times as we go from child to parent and so the total number of tree operations is bounded by $\mathcal{O}((k/z)^d \log_z k)$. If the node is the root of an exhausted merger, we may bring no elements to the parent. However, this can only happen once per node for a total of k times. □

4.1.1 The Sorting Algorithm

Now that we are able to merge multiple sorted streams cache-obliviously and efficiently, we can use that in a mergesort algorithm. The question is now, what order funnel should be used for the merging? We may simply use $k = N$, in which case the sorting algorithm *is* the merging. However doing so we are only feeding the merger one element per stream, and that would violate the requirement that a total of at least αk^d elements is in the input, since d is a constant greater than one. Furthermore, the merger would take up a super-linear amount of space. The next obvious choice would then be $k = (N/\alpha)^{1/d}$ corresponding to the funnel outputting exactly N elements, in which case there are enough elements in the input to satisfy the funnel requirements and the funnel will require sub-linear space as well. The algorithm proceeds as outlined in Algorithm 4-5.

Algorithm 4-5. funnelsort(Array A)

```
if |A| ≤ αz^d then
    sort A "manually"
else
    split A into roughly equal-sized arrays S_i, 0 ≤ i < (|A|/α)^(1/d)
    for 0 ≤ i < (|A|/α)^(1/d)
        funnelsort(S_i)
    construct a (|A|/α)^(1/d)-funnel F
    attach each S_i to F
    warmup(F.root)
    fill(F.root)
fi
```

It is understood that both funnelsort and fill require some way of providing the output. This could be in the form of a pointer to where the first (smallest) element should be placed. The output of fill will be the same as the output of funnelsort, but discussions on exactly where the output of funnelsort will go, is deferred to the next chapter.

The base case threshold is set to αz^d, simply because it needs to be set somewhere. This is a nice place, since it guaranties, that $k > z$, when a k-funnel is used, which may eliminate the need to handle some special cases in the funnel. Exactly how the manual sorting is done, whether with quicksort or bubblesort or something third, is asymptotically insignificant, since it done on a constant-sized array.

Theorem 4-3. Assuming $M^{(d-1)/(d+1)} \geq cB$, for some $c > 0$, Algorithm 4-5 performs $\mathcal{O}(N \log N)$ operations and incurs $\mathcal{O}(dN/B \log_M(N))$ memory transfers to sort an array of size N, for fixed α and z.

Proof. The base case is when all elements fit in cache twice, once for the input and once for the output, along with the funnel needed to merge. In that case, funnelsort incur at most the memory transfers needed to fetch the input into cache. If the input, output and funnel does not fit cache, the second property of the funnel, states that αk^d is $\Omega(kB)$. Further more, with $k = (n/\alpha)^{1/d}$, Theorem 4-1 states that merging n elements incur $\mathcal{O}(\log_M(n)/B)$ memory transfers per element. This gives us the following recurrence for the number of memory transfers incurred by Algorithm 4-5:

$$T(n) = \begin{cases} \mathcal{O}\left(n/B\right), 2n + S\left(\left\lceil n/\alpha^{1/d} \right\rceil\right) \leq M \\ \left\lceil n/\alpha^{1/d} \right\rceil T\left(\left\lceil \alpha^{1/d} n^{\frac{d-1}{d}} \right\rceil\right) + \mathcal{O}\left(n/B \log_M n\right), otherwise \end{cases} \tag{4.14}$$

The recurrence expands to the sum

$$
\begin{aligned}
T(N) &= \mathcal{O}\left(\sum_{i=0}^{\log\log_M N} N^{\left(\frac{d-1}{d}\right)^i} \left(\log_M N^{\left(\frac{d-1}{d}\right)^i}\right) B^{-1}\right) \\
&= \mathcal{O}\left(N/B \log_M N \sum_{i=0}^{\log\log_M N} \left(\frac{d-1}{d}\right)^i\right) \\
&= \mathcal{O}\left(d\, N/B \log_M N\right)
\end{aligned}
\tag{4.15}
$$

For the work bound, we know from Theorem 4-2, that the work done in the funnel is bounded by the number of moves. Since funnelsort, when unrolled, is essentially one big funnel, all elements are merged through $\mathcal{O}(z\log_z N)$ base mergers for a total of $\mathcal{O}(N\log N)$ comparisons and moves, for fixed z. □

Using (3.10) and Theorem 4-2, we conclude that funnelsort is an optimal cache-oblivious sorting algorithm.

As with multiway mergesort, we can see that as long as $\alpha^{1/d}N^{d-1/d} < M$, only the top level of recursion goes outside the cache, incurring at most $4N/B+4$ memory transfers, including those that write back blocks. With M half a gigabyte, $\alpha = 1$, and $d = 2$, this will be the case for all inputs taking up less than 2^{58} bytes, thus we do not expect funnelsort to perform any different from multiway mergesort on this particular level of the memory hierarchy.

4.2 LOWSCOSA

When it comes to the practical application of algorithms minded for massive data sets, it is not only the asymptotic I/O complexity, but also the amount of memory actually needed to perform the task, that is of importance. The space required by the algorithm, can be divided into the space required to store the actual input and workspace. Clearly, a sorting algorithm needs linear space to store the input, and indeed most external memory algorithms require $\mathcal{O}(N)$ space in total. Many, however, also require linear workspace.

Multiway mergesort, unfortunately, is one such an algorithm. While it has good, indeed asymptotically optimal I/O performance, it requires space the size of the array being sorted to store the element in sorted order. These elements are mere duplicates of the elements in the input, so keeping the input around seems wasteful, yet it is necessary for the algorithm to work. Being able to work with data, using very little extra space would be a benefit; compared to otherwise equal sorting algorithms, mergesort might incur up to *twice* the memory transfers of an in-place variant, when $M < N$, simply because we need not incur memory transfers writing the output to previously untouched memory. When $M/2 < N \leq M$, it is even worse; sorting using an algorithm like funnelsort would incur N-$M/2$ memory transfers to write the output, not counting the ones needed to bring the elements into cache, while in-place algorithm would incur no memory transfers at all. In this section, we present a novel extension to k-mergers that enable them to be used in a sorting algorithm that does not require linear working space, in a way that does not affect its overall I/O complexity.

4.2.1 Refilling

The idea for a cache-oblivious sub-linear workspace merge-based sorting algorithm is quite simple; the merger is extended with a refilling feature. The motive is to recycle the space occupied by elements that have already been or are in the process of being merged, while maintain temporal and special locality.

The technique can be applied to heap-based mergers as well as funnels. It consists of requiring the merger to invoke a predefined function when elements have been read in from the input. The function is given an indication of from what section of what input stream they were read. Furthermore, the merger must invoke it before the elements read in is written to the output. The function knows that the elements from that part of the input now is in the hands of the merger, and will eventually be output, so it is free to reuse the space they take up. A key to making this work is that the merger is obligated to invoke this function after having read in no more than a constant number of elements. This way, the I/O complexity of the merger is unaffected; that section of the input is in cache because it has recently been read by the merger. In case of the funnel, one would invoke the refiller after having called fill on a leaf. At this point, at most αz^d elements have been read in from a given stream, so, assuming $2\alpha z^{d+1} < M$, that part of the stream is still in cache, and it can be recycled for free, or at least the price of fetching the elements used. The refilling functionality may in general be useful many scenarios, where space is sparse.

4.2.2 Sorting

The basic steps for the low-order working space cache-oblivious sorting algorithm (LOWSCOSA) is first to provide for a “working area” inside the input array. When that is done, the merger will then sort recursively and then merge using the working area. What we will do is partition the array into two equal sized parts, the first containing elements larger than all elements in the second, then recursively sort the second part. The first part then becomes the output of the merging of the second, but with the merger refilling what it reads in with elements from the first part. The process is illustrated in Figure 4-3. The refiller maintains a pointer to elements in the first part of the array, initially pointing to the first, going forward from there. When invoked, with a section of size n, it copies n elements beginning from the pointer to that section.

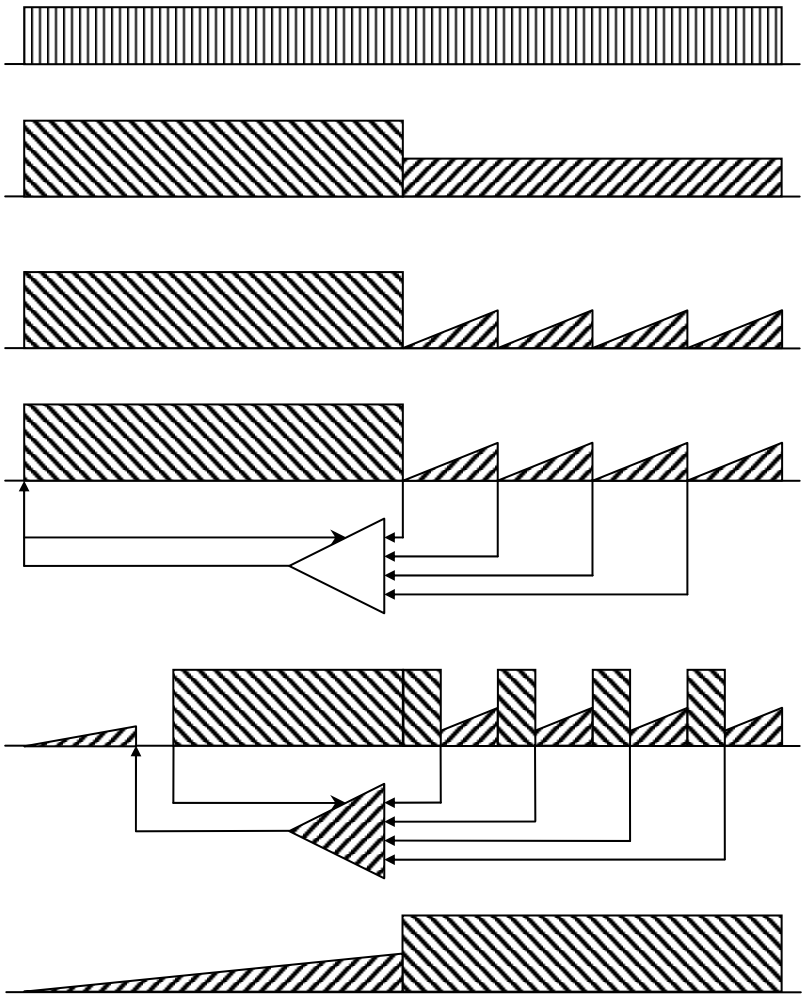


Figure 4-3. The process of multiway merging with low-order working space. Horizontal sections indicate unordered elements. First step partitions the array into large (diagonal-down patterned) and small (diagonal-up patterned) elements. Arrowheads indicate flow direction of elements; the arrow going into the side of the merger is the refiller. The gap between the output and the refiller shows that elements are read in through the refiller before elements are output from the merger.

Since the refiller is invoked before elements are output, the refiller pointer will always stays ahead of where the output is written, and within the first part of the array. The algorithm has now sorted the first half of the array. It then recurses on the second half, as is illustrated in Algorithm 4-6, where a funnel is used as merger. The merger and the parameters can be changed to that of a cache-aware mergesort.

Algorithm 4-6. LOWSCOSA(Array A)

```
if 2|A| ≤ αz^d then
    sort A "manually"
else
    partition A into a half of large elemens A_l and a half of small A_s
    split A_l into roughly equal-sized arrays S_i, 0 ≤ i < (N/(2α))^(1/d)
    for 0 ≤ i < (|A|/(2α))^(1/d)
        funnelsort(S_i)
    construct a (|A|/(2α))^(1/d)-funnel F
    attach each S_i to F
    set pointer in F.refiller to A_l
    warmup(F.root) // A_s now contain large elements
    LOWSCOSA(A_s)
fi
```

We will discuss the practicality of the partitioning in Section 5.5.2. For now, in the analysis, we use of the following lemma:

Lemma 4-2. Finding the median of an array incurs $\mathcal{O}(1+N/B)$ memory transfers, provided $M \geq 3B$.

Proof. The proof can be found in [Dem02]. Since the algorithm relies on constant time operations and scanning, the result is rather intuitive. □

With the median at hand, we can make a perfect partitioning efficiently, thus avoiding any complications and give a worst-case bound.

Theorem 4-4. Assuming $M^{(d-1)/(d+1)} \geq cB$, for some $c > 0$, and $M > 2\alpha z^{d+1}$, Algorithm 4-6 performs $\mathcal{O}(N\log N)$ operations and incurs $\mathcal{O}(dN/B\log_M(N))$ memory transfers to sort an array of size N.

Proof. Elements now pass through the merger in two ways; either through the refiller or through input streams being merged. After the partitioning the large elements pass through the refiller, incurring $\mathcal{O}(B^{-1})$ memory transfers each. The small elements gets sorted recursively incurring $\mathcal{O}(d\log_M((N/\alpha)^{1/d}/2)/B) = \mathcal{O}(d\log_M(N)/B)$ memory transfers each, pass through the merger, incurring $\mathcal{O}(\log_M(N)/B)$, but is not present at the next level of recursion. The base case is when all elements and the merger used to merge the recursively sorted streams fits in cache. From then on, no memory transfers are incurred. Using Theorem 4-1, Theorem 4-3, and Lemma 4-2, we get the recurrence

$$T(n) = \begin{cases} \mathcal{O}(1), n + S\left(\left\lceil n/\alpha^{1/d}\right\rceil\right) \leq M \\ \mathcal{O}\left(n/B\right) + \mathcal{O}\left(d\,\frac{n}{B}\log_M n\right) + T\left(\left\lceil n/2\right\rceil\right), otherwise \end{cases} \tag{4.16}$$

which expands to

$$\begin{aligned} T(N) &= \mathcal{O}\left(\sum_{i=0}^{\log N} 2^{-i} N \left(1 + d \log_M \left(2^{-i} N\right)\right) B^{-1}\right) \\ &= \mathcal{O}\left(B^{-1} d \log_M N \sum_{i=0}^{\log N} 2^{-i} N\right) \\ &= \mathcal{O}\left(d \, {}^{N}\!/_{B} \log_M N\right) \end{aligned} \tag{4.17}$$

The work bound is given by a similar recurrence:

$$\begin{aligned} T(n) &= \begin{cases} \mathcal{O}(1), 2n \leq \alpha z^d \\ \mathcal{O}(n) + \mathcal{O}\left({}^{n}\!/_{2} \log n\right) + T\left(\left\lceil {}^{n}\!/_{2} \right\rceil\right), otherwise \end{cases} \\ &= \mathcal{O}\left(\sum_{i=0}^{\log N} 2^{-i} n \left(1 + \log\left(2^{-i} n\right)\right)\right) \\ &= \mathcal{O}(n \log n) \end{aligned} \tag{4.18}$$

□

So the LOWSCOSA is also optimal in the cache-oblivious model, but what is unique to it is that the following theorem holds:

Theorem 4-5. Algorithm 4-6 requires no more than $\mathcal{O}(N^{(d+1)/(2d)}+N^{(d-1)/d})$ working space.

Proof. Working space is required to store the funnel and the output of the sort of . Since the algorithm is tail-recursive, it requires no stack to store the state of the recursive calls. At each level, we use funnelsort to sort subarrays of size $\mathcal{O}(N^{(d-1)/d})$. We allocate space to store the output of one call to funnelsort; when the sort is completed, the elements can be copied back and the allocated space reused in the next call to funnelsort. When we are done sorting using funnelsort, the space is freed and space allocated for the funnel. From the proof of Lemma 4-1, we have that, a k-funnel takes up $\mathcal{O}(k^{(d+1)/2})$ space. With $k = \mathcal{O}(N^{1/d})$ that becomes $\mathcal{O}(N^{(d+1)/(2d)})$. For $1 < d \leq 3$, the space required for the funnel dominates while for $d > 3$, space requirements are bounded by output of the funnelsort. Both terms are bounded by $\mathcal{O}(N^{(d+1)/(2d)}+N^{(d-1)/d})$, which is $o(N)$ for all $d > 1$. Before continuing recursively on the second half, all workspace is freed. This way the total working space consumption is maximal at the top-level of the recursion. □

Chapter 5

Engineering the Algorithms

Having presented the algorithms in a theoretical setting, it is now time to start looking at them from a practical perspective. In this chapter, we do a thorough investigation of what can be done to maximize performance of the algorithms presented in the previous chapter.

We will look into the design choices left open in the previous chapter and fill in the details while following a path leading to an implementation of high performance. In mergesort algorithms, in general we cannot avoid doing $N \log N$ comparisons and $4N/B$ I/Os. The mergesort algorithms we investigate also achieve this, so maximizing performance largely amounts to minimizing overhead. In this chapter, we will focus on possible approaches to minimizing overhead and evaluate these approaches through experimental analysis.

In the next chapter, we will compare our implementation to that of other algorithms. For that, it is important to ensure we have a reasonable efficient implementation [Joh01]. The results presented in this chapter provide knowledge of what combination of parameters and algorithmic details yield fast algorithms and data structures. This knowledge is then combined into optimized cache-oblivious sorting algorithms, the performance of which will be evaluated in the next chapter.

We begin this chapter with a section with general considerations on how we evaluate performance of algorithms. The remaining chapter is then dedicated to the implementation of the algorithms. In Section 5.2, we provide an overview of the structure of and the pieces that make up the implementation. In Section 5.3, we look into aspects of the funnel data structure. Of particular interest in this section is how to manage the data structure and how to implement a high performing fill algorithm. Section 5.3.4 focuses on the funnelsort algorithm. We provide a few optimizations and investigate good parameters for determining subproblem sizes and buffer sizes of the funnel. Section 5.3.4 provides a discussion on implementation details of the LOWSCOSA.

5.1 Measuring Performance

The focus of this and the next chapter will be on performance evaluation. Before presenting any benchmarks, we want to make clear exactly what it is we will be showing. The following is an overview of how the benchmarks, presented in both this and the next chapter, are conducted.

5.1.1 Programming Language

For portability and more importantly genericity, the algorithms are implemented in a generic high-level language. The language chosen is C++ [C++98]. The primary reason for this is that it allows for producing high performing code, while implementing generic algorithms. C++ was designed to have optimal run-time efficiency; depending on compiler quality, abstraction penalties are minimal. In addition, the accompanying library, the Standard Template Library (STL), contains a highly optimized sorting function, named `std::sort`, with which we may compare the algorithms developed here.

5.1.2 Benchmark Platforms

The underlying platforms for the benchmarks have been chosen based on diversity and availability. As discussed in Chapter 2, different processors and operating systems behave and perform different under certain circumstances. It is thus important to cover as many types of processors as possible, when arguing that the design choices made will be sound on not one but many different architectures. Our implementation would only be compelling to people using one particular platform, were we only able to show high performance on that type of platform.

We feel it is important to benchmark in real world scenarios and have thus chosen not to use simulation tools and to use the memory subsystem as is; we will not reduce the memory available to our algorithms artificially. The fact that modern computers now come with at least half a gigabyte RAM makes results obtained on machines artificially restricted to 32 or 64 MB of RAM of no practical relevance.

Hardware

Benchmarks made on three radically different architectures to ensure that we do not accidentally tune the algorithms for a specific architecture, thus defying one of the design goals of cache-oblivious algorithms. The architectures are Pentium 3, Pentium 4 and MIPS 10000 based. Their specifications can be found in Appendix B.

The Pentium 3 platform represents the traditional modern CISC. It has a pipelined, out-of-order, and super-scalar core. Its pipeline is as short (12 stages) as seem sensible when designing CISCs. The Pentium 2 and the AMD Athlon both have designs similar to the Pentium 3 and is thus expected to perform comparatively. The Pentium 4 computers represent a significant change in design philosophy. They signify a departure from the ideal of keeping pipelines short to minimize the cost of pipeline hazards and feature a 20-stage pipeline. This means that a branch miss-predict may waste as much as 20 clock cycles. The benefit of the long pipeline is very high clock rates. In applications such as sorting, where unpredictable conditional branches are commonplace, a 20-stage pipeline may well cause performance to degrade despite the high clock rate. To counteract the performance loss due to branch miss-predicts, the Pentium 4 employs the most sophisticated branch prediction logic of the three processors. Whether it will help it in the context of sorting remains to be seen.

To represent the RISC family of processors, we include a MIPS 10000 based computer. It has a traditional 6-stage pipeline and the simplicity of the core has made the inclusion of a large 1MB L2 cache possible. A notable feature of this processor is its ability to use an address space larger than 2^{32} bytes. Its word size is 64 bits both when used as address operation operands, and in the ALU. It is a relatively old

processor and it is significantly slower than the Pentiums, so unfortunately due to time constraints, it could not participate in all benchmarks, though we have included it in all benchmarks presented in this chapter to guarantee that our implementation is not optimized for the Pentiums alone.

We feel that these three platforms are representative of most computers in use today in that most processors in use to day have a design resembling one of these three CPUs.

Software

On the software side, the Pentium computers are running the Linux operating system and the MIPS computers are running the IRIX operating system. The primary development platform, however, has been Windows. We feel that this has also contributed to diversifying the code.

The compilers used are listed in Appendix B. All executables used to generate benchmark results were compiled using the GNU Compiler Collection (GCC), which is the only one available on all platforms used. This was done to ensure that no algorithm had the benefit or detriment of good or poor code generation from the compiler, on any of the platforms. Say, for example, the MIPS Pro compiler is very good at generating code for funnelsort and not for `std::sort`. This would then put aspects of `std::sort` in a particular bad light, but only on the IRIX platform. We have found that for our experiments, the GCC generates code that is at least as good as any of the other compilers used generates. If anything, it generates very fast code for the quicksort implementation included in the standard library. The code generated by the MIPS Pro compiler was of equal quality, but both the Intel and Microsoft compilers generated significantly slower code.

5.1.3 Data Types

Sorting is used in a wide variety of applications. It is important that our benchmarks closely reflect as many applications as possible [Joh01]. Recent efforts in developing in particular cache-efficient sorting algorithms have opted to evaluate the performance sorting elements consisting only of a single integer key ([LL99], [XZK00], and [ACV+00]). We feel, however, that sorting only integers is of limited applicability; some sort of information should be associated with the integers. At the very least, a pointer to some structure should accompany the integer. This may have an impact on algorithms that move elements a lot.

Inspired by the Datamation Benchmark, in turn inspired by sorting problems encountered in the database community, we have included a type of size 100 bytes [DB03]. The problem in the Datamation Benchmark originally consisted of sorting one million such records. This has since proved too easy and the total time became dominated by startup time. In response, the problem was changed to that of sorting as many records in one minute. This is known as the Minute Sort Benchmark. Since there are no restrictions on the platforms used when performing the benchmark, these benchmarks are largely a test of hardware and operating system I/O subsystems, rather than algorithm implementation. To allow contenders with limited finances to compete, the Penny Sort Benchmark was introduced. This benchmark is essentially the Minute Sort Benchmark with the result scaled by the price of the platform used in dollars. Algorithms competing in this benchmark are however still designed to be fast on one specific platform.

For our benchmarks, we choose to look at these three data types:

- Integers. This data type is simply a `long`.
- pairs. These represent key-value pairs and are implemented as class with data members of type `long` and `void*`. Their relative order is based on the value of the `long`.
- records. Represents database records or equivalents. They are implemented a class with a data member of type `char[100]`. Their relative order is determined by the `strncmp` function of the C standard library, such that the entire record is also the key.

Note that on the MIPS machine, both `long` and `void*` are 64-bit, while they are 32-bit on the Pentiums.

It would be infeasible to conduct the entire study in this chapter with several different data types. Thus in this chapter we will only use pairs. We then risk optimizing for relatively small data types. We will be weary of this when it comes to choosing between implementations that favor small elements, and then evaluate the performance of our implementation used on all three data types in the next chapter.

For the same reason, in this chapter, we limit the experiments to uniformly distributed random data. In addition, we do not want to optimize for any special case distribution, and since some results could be highly dependant on the distribution of elements and our algorithm implementation should not favor any distribution, we conduct the experiment on uniformly distributed pairs.

To generate random keys, we use the `drand48` function available on both Linux and IRIX. In Windows, we use the `rand` function of the C standard library.

5.1.4 Performance Metrics

We may measure performance of our implementations by several ways. Here we bring an overview of the metrics used in this thesis.

Running Time

Of absolute primary concern is the total time spending solving the problem (sorting, merging, or other) measured on a physical clock. This measure is an indication of how long one would wait for the problem to be solved, which we believe to be of primary concern to the user of our algorithms.

As an alternative to the wall clock time, one may use the *CPU time*. That is the total time the algorithm is actually running on the processor. This measure is important if we were to estimate how much the processor would be occupied by the solving the problem. This could be of concern when other processes need access to the CPU. However, our implementation will not be designed with multiprocessing in mind. Furthermore, measuring the CPU time does not take the time the algorithm spends waiting for a page fault into account, because during this time, it is not scheduled on the CPU. Thus, we will not consider CPU time for our benchmarks.

The *wall clock time* is determined using the `gettimeofday` C library function. In Linux and IRIX, it appears to have a precision in the order of microseconds. In Windows, it appears to have a precision of milliseconds only, so we use the high-resolution

performance counter available through the QueryPerformanceCounter API. This appears to have a precision of a couple of nanoseconds.

Page Faults
Albeit not of primary concern, the number of page faults incurred running the algorithm may provide us with important insights into the behavior of the total running time of the algorithm.

In the next chapter, when sorting large data sets, we will thus also present the number of page faults incurred by the algorithms. For this, we use the get_rusage system call in Linux and IRIX. This call provides both the number of minor and major page faults. Since only the major page faults have significant impact on performance, on that number will be reported. In Windows, we assign a job object to the process and use the QueryInformationJobObject API.

Cache and TLB misses
Aside from the number of page faults, the number of cache and TLB misses also influence performance. Performance Application Programming Interface (PAPI) allows for monitoring hardware counters [PAPI03]. Hardware counters can keep track of cache misses, TLB misses, and similar hardware events. PAPI is a cross-platform software library that provides access to these counters. Unfortunately, to make the Linux version work, a patch has to be applied to the kernel and we did not have that privilege for our test machines.

No patch was needed for the IRIX version, however, so we can use PAPI on the MIPS machine to show the cache behavior of the algorithms. We will be measuring the number of L2 cache misses (the PAPI_L2_DCM event) and TLB misses (PAPI_TLB_TL).

5.1.5 Validity

To provide valid and relevant experimental analysis, we should attempt to even out any disturbances in the results due to effects external to the algorithm, such as the scheduling of other processes running on the system. This may be achieved through more or less elaborate ways of averaging results from multiple runs of the same algorithm on the same problem.

In this thesis, however, we deal with such massive data sets, that individual benchmark runs take several minuets, sometimes even several hours. In comparison, any anomalies due to process scheduling or other operating system operations often cause no more than in the order of milliseconds of delays in the running time, so we do not expect this to influence our results greatly. Periodic scheduling of other processes may interfere significantly with the result of the benchmark. However, such interference is only normal in modern multiprocessing environments.

Primarily for time considerations, we choose to run each benchmark only once. This means that sudden “jumps” in measurements may be present in the results. However, we will attempt to run them with as many different parameters as feasible, to expose any systematic behavior of interest, and to expose what may be irregularities and what reflects actual algorithm performance.

5.1.6 Presenting the Results

A vast number of benchmarks have been run. Not all of them present new and relevant information. To avoid cluttering the discussion in the present chapter, the results of all benchmark are included in Appendix C and in electronic form as described in Appendix A, and only the ones that present relevant information will be included in the text. For the rest of the results, we thus refer to Appendix A. The results are presented in **Chart**s and their number in the appendix corresponds to their number in this chapter. The titles of the charts are consistently titled <processor>, <cache size>/<RAM size> with <...> substituted with values of the machine they were run.

The engineering effort carried out in this chapter is intended to compare different approaches to solving the same basic problems; they should not be viewed as performance evaluations of the individual implementations. Thus, when comparing an algorithm using method A with B and C, we prefer to show the performance of method B and C *relative* to A. This is done to emphasize what is the focus of this chapter, namely identifying approaches to implementing the algorithms that maximize performance. A more absolute performance analysis is carried out in the next chapter, when we have found the best way to implement the algorithms.

For each benchmark, we discuss exactly which part of the algorithm we will be analyzing. Each benchmark is accompanied by a discussion of the results, relating them to design choices made. Based on this we draw conclusion on what choices result in efficient implementation solving the problem.

5.1.7 Engineering Effort Evaluation

The engineering effort presented in this chapter seeks to find a good way to implement the algorithms. To do this, a series of questions need to be answered, such as

- How should the funnel be laid out in memory?
- How do we locate nodes and buffers in the funnel?
- How should we implement the merge functionality?
- What is a good value for z and how do we merge multiple streams efficiently?
- How do we reduce the overhead in the sorting algorithm?
- How do we sort at the base of the recursion?
- What are good functions for determining output buffer sizes and sizes of subproblems to recurse on in the sorting algorithm, i.e. what are good values for α and d?

All of these questions have multiple possible answers that will influence the performance of our implementation. The answer to one question does not necessarily influence the answer to another. Finding the answer to all questions that combine to yield an optimally performing implementation implies searching the entire space of possible combinations of answers. This space is so vast it would simply be infeasible. What we thus do in this chapter is examine one question at a time, first determining the best layout of the funnel, then the best way to locate nodes, and so on. We suspect that the result of this investigating the design options in this manner will bring us very close to an optimally performing implementation.

Since α and d influence both the funnel data structure and the funnelsort algorithm, we postpone the analysis of what constitute good values until the implementation

details have been settled. Until then, we do not know what values will yield a fast sorting algorithm, so we choose by intuition. As a guideline, we choose large values when analyzing choices that influence merging, such as how to implement binary merging, and large values for choices that concern the tree structure, such as layout of the funnel. Since small values will yield small buffers and thus fewer elements merged per node we visit, we will expose aspects of the performance relating to operating the funnel. Conversely, large values will yield large buffers and likely more elements merged per node we visit, thus emphasizing the performance of the implementation of the merging algorithm. Regardless of the choice of values for the constants, for consistency we merge k streams of k^2 elements. This may not be the ideal for all values of the constants, but it is necessary to compare across different values of constants, since merging $k' < k$ streams is easier than merging k streams.

When measuring performance of the funnel (Section 5.3) we do not store the output of the funnel, we only check that the elements are output in sorted order. We do this to eliminate the overhead of writing and storing all the elements. Since this overhead is common for all implementations of the funnel, it does not influence a study comparing different implementations. As an added benefit, we automatically verify that the result of the algorithm is correct.

5.2 Implementation Structure

In this section, we give an overview of the pieces that make up the implementation, how they relate to each other, and what their roles are. We provide illustrative interfaces and defer the implementation details to the following sections.

5.2.1 Iterators

The concept of iterators is used extensively in the STL. An iterator has the functionality of a traditional pointer, in that it can be dereferenced to give the object it points to. As with pointers to elements in arrays, an iterator can also be incremented to point to the next element. However, any class with these properties is an iterator, so iterators serve as a generalization of the traditional pointer and its relationship with the array; they represent a general way of iterating through the elements of a data structure (*container* of elements) in essence, a way of flattening the structure.

Using iterators is the primary way of implementing generic algorithms in C++. The algorithm is designed without any knowledge of with what type of iterator it is used. By this token, we can implement any container of elements and have an algorithm work on it by implementing an iterator for it. In that sense, iterators are the glue that binds together algorithms and elements. By abstracting away the implementation of the iterator from both the containers and the algorithm, any algorithm can be made to work with any set of elements.

Some containers are not as easy to navigate as arrays. For instance, one cannot (at least in constant time) add, say, twenty to an iterator pointing to an element in a linked list and get an iterator pointing to the element twenty past the original. For this reason, the STL defines six categories of iterators, by what operations can be done on them in constant time:

- *Input iterator*. An iterator that can only be dereferenced, incremented and compared for equality.
- *Output iterator*. An iterator that can only be dereferenced and incremented. The result of a dereference must be assignable, that is, the expression `*x = t, ++x` must be valid for some object `t` if `x` is an output iterator. Equality comparisons are not requred by output iterators.
- *Forward iterator*. An iterator that can be dereferenced and incremented. Further, an iterator can be compared with other iterators to determine the relative positions of elements they point to. The result of a dereference should be a reference to an object, as opposed to output iterators that are allowed to return proxy object to which objects can be assigned.
- *Backward iterator*. Same as a forward iterator, except it can be decremented, not incremented.
- *Bidirectional iterator*. An iterator that is both a forward and a backward iterator.
- *Random access iterator*. An iterator with all the functionality of a traditional pointer; a distance between two elements can be computed and integer arithmetic can be done on it and iterators can be advanced a given distance. A random iterator is also a bidirectional iterator.

A goal when designing algorithms is to restrict the requirement of the iterators used, as much as possible.

We will be using the iterator abstraction throughout the implementation.

5.2.2 Streams

A stream is a sequence of elements. Its state consists of where to find the next element and how many remain. To this end, we simply represent streams as a pair of input iterators, one that points to the next element, and one that points one past the last element. A stream is constructed from two such iterators. The iterator pointing to the next element is returned by the member function `begin` and the iterator pointing the one past the last element is returned by `end`.

Most containers implemented in the STL, such as `std::vector`, `std::list`, and `std::set`, have member functions `begin` and `end` with the same semantics. Streams can thus be used as wrappers for any of these containers, as well as ordinary arrays. Streams may also be used to represent continuous (in the sense of the iterator) subsets of the containers, in essence slices of the flattened data structure.

5.2.3 Mergers

The STL provides a binary merge algorithm. It is declared as

```
template<class InIt1, class InIt2, class OutIt>
OutIt merge(InIt1 begin1, InIt1 end1, InIt2 begin2, InIt2 end2, OutIt dest);
```

where the `InIt` name indicates that it only requires `begin1`, `end1`, `begin2`, and `end2` to be input iterators and the `OutIt` indicates that `dest` should at least be an output iterator. The precondition is that there are a sorted set of elements between `begin1` and `end1`, and between `begin2` and `end2`. When the function returns, all elements of these sets have been written consecutively to `dest`, by dereferencing, assigning, and incrementing.

For merging in our implementation, a somewhat different approach is needed. First, we may want to merge more than two streams at a time. Secondly, a different set of semantics is needed. We need two kinds of mergers: the general merger and the basic merger. Their semantics differ and their interfaces reflect it, yet they are similar.

To accommodate for more than two input streams, both are implemented as function objects rather than functions. Function objects are simply objects that can be used as functions. As any object, they maintain a *state*. The input streams of a merger are then a part of the state of the function object, allowing us to add input streams to the merge function. With the merge interface of the STL, we are restricted by the number of arguments we can provide; however, there is no language restriction on the number of times, we can alter the state of a function object.

The basic merger has a compiletime set limit on the number of streams it can merge. Between zero and that limit of streams can be associated with it. Empty streams cannot be associated with a basic merger. Attempting to do so will have no effect. The semantics is essentially that of the Algorithm 4-4 on page 48; it merges as long as there is room in the output and elements in all associated streams. The associated streams are updated to reflect that elements have been extracted. To pass on information about which stream caused the merger to stop by becoming empty, we use a concept of tokens. When a stream is associated with the basic merger, a token is in turn associated with the stream and when invoked, the basic merger is given the output and what token to associate with the output. When done merging, it simply returns the token associated with the stream that caused it to stop. The interface looks like this:

```
template<int Order, class InStream, class Token>
class basic_merger
{
public:
    typedef Token token;
    void add_stream(InStream *s, token t);
    template<class FwIt>
    token operator()(FwIt& dest, FwIt dest_end, token outtoken);
};
```

Note that tokens can be anything from a simple integer indicating the number of the stream or a pointer to a complex user defined object. They are expected to be small, however. Note also that the first argument of the operator() is passed by reference, so it too can be updated. We require forward iterators because we need to be able to compare them to see if we have hit the end of the input. Order is the order of the basic_merger, also denoted z. If add_stream is called more than Order times with non-empty streams the state of the merger becomes undefined, as is the state after a merger has been invoked. As a simple illustration, here is what Algorithm 4-4 could look like using a basic merger:

```
template<class Node>
void fill(Node *n)
{
    basic_merger<2,typename Node::stream,Node*> merger;
    merger.add_stream(n->left_input, n->left_child);
    merger.add_stream(n->right_input, n->right_child);
    n = merger(n->out_begin, n->out_end, NULL);
    if( n )
        fill(n);
}
```

In this example, we use `Node*` as tokens. The right input buffer is associated with the right child and the same with left. For the output token, we simply use `NULL`, so if the merger returns non-null, we call recursively on the node returned.

General mergers will be used on a larger scale and should thus provide for an arbitrary number of input streams. The semantics differ from the basic merger either in that it merges until the output is full or until *all* input streams are empty. This eliminates the need for tokens. The interface looks like this:

```
template<class InStream, class Refiller, class Allocator>
class general_merger
{
public:
    general_merger(int order);
    general_merger(int order, const Allocator& a);
    static typename Allocator::size_type size_of(int order);
    void add_stream(const InStream& s);
    template<class OutIt>
    OutIt operator()(OutIt dest, OutIt dest_end);
    template<class OutIt>
    OutIt empty(OutIt dest, OutIt dest_end);
    void reset();
    void set_refiller(const Refiller& r);
    const Refiller& get_refiller();
    stream_iterator begin();
    stream_iterator end();
};
```

Among the main differences are that streams are now copied and maintained internally; there is no obligation to maintain associated streams. It is still possible to see how far the streams have advanced, by running through them using the `stream_iterators` returned by `begin` and `end`. It is possible to invoke the merger repeatedly. The `empty` member function template is there in anticipation of the merger storing elements internally after they have been read from the input and before they are written to the output. `empty` then provides a way of retrieving these elements, in no particular order. The `reset` member function sets the begin iterator of each stream to the end, essentially marking them all empty, and resets the internal state of the merger. This has the effect of destroying the merger and creating a new one with the same order. The `get_refiller` and `set_refiller` provides the interface for adding a refiller as described in Section 4.2.1. The rest of the interface has to do with memory management, to which we will return.

5.3 Funnel

We choose to implement the two-phase funnel, not because it is the easiest to implement, but because the simplicity it brings to prior funnel variants will not make it perform any worse and likely make it perform better, as discussed in Section 4.1.1. The funnel is a k-merger and when input streams are added, its implementation as such follows the interface of a general_merger.

5.3.1 Merge Tree

We will denote the combined funnel and input streams a *merge tree*. A merge tree consists of nodes and buffers. Buffers can contain any number of elements. These elements can only exist contiguously in the buffer, but they need not be located at the tail or the end of the buffer. When calling fill on a node, we need to identify where the elements are in the buffer, so we can resume from where we left off the last time we were filling its output buffer. A minimal description of the state of the merger is thus a pair of iterators for each buffer. Conceptually, a node may also contain pointers to where the buffers begin and end, as well as its parent and its children.

While a node need not maintain pointers for both its input and output buffers, it should do so for either the inputs or the output. Which one is not clear; a natural one to one relation ship exist between a node and its output, however, if we have no information about the state of the input buffers of a node, we have to go to the child nodes to get it, when we first start filling. This can reduce locality of reference, since nodes are not generally located near their children. We consider that an important aspect, so in our implementation we choose to let a node be responsible for the state of its input buffers. Figure 5-1 illustrates a node (with $z = 2$), the triangle, before fill returns from its right child. Also depicted are four pointers per buffer. Head (h) and tail (t) indicate the beginning and end of the contiguous section of elements in the buffers and begin (b) and end (e) indicate the beginning and end of the entire buffer.

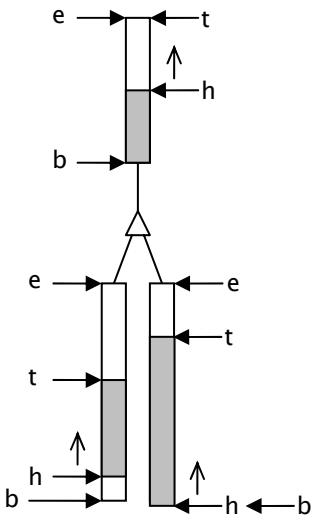


Figure 5-1. The pointers involved in a fill operation.

The situation in the figure is that prior to invoking its right child, fill called fill on the left child. That fill operation caused the subtree to become exhausted and thus the output was not completely filled. Some elements were then merged from the left input before the right got empty and fill was invoked on the right child. It, in turn, exhausted its subtree before returning.

Basic mergers are used to carry out the fill. Before invoking the basic merger, streams consisting of the head and tail of the input buffers are added to it, using add_stream. Then it is invoked with head and tail of the output as its arguments. This requires an invariant that *elements in input streams lie from head to tail and elements in the output stream lie between begin and head*. To maintain this invariant, we *flip* the buffers with a flip operation as we pass them when calling recursively on a child node or return from a recursive call. It consists of the double assignment (t = h, h = b). When returning from a fill, by induction, we know that the buffer we passed contain elements from b to h. After the flipping the buffer, we have h equal to the old b, the beginning of the elements, and t equal to h, the end of the elements, and thus a valid input buffer. Conversely, when calling recursively and passing an input buffer the flip operation turns the buffer into a valid output buffer.

As discussed, the gereral merger interface allows for arbitrary types of input streams, while the funnel maintains its own buffers. These buffers are elements allocated from the heap and the iterators used when merging them are simple pointers stored in the nodes. However, the input streams of the general merger cannot in general be represented by a pair of pointers. This presents two problems. First, we are wasting space storing pointers we are not using. Second, the leaves do not readily know from where to get the input. The first problem is easily solved by not actually allocating space for the leaf nodes.[1] We may then say that pointers the non-existing leaves in their parents are wasteful, however they are not, because we need some way to distinguish internal nodes from leaf nodes. The second problem is solved by storing the input streams in a separate array. When calling recursively, we keep track of the path we took and use it to locate the appropriate streams in the array. This in turn will give a minor overhead, however we consider it a small price to pay to get genericity.

5.3.2 Layout

As early as 1964, laying out trees in a particular way was known to be useful; careful layout of the tree used in the implementation of the heap, paved the way for the in-place heapsort [Wil64]. For our purpose, neither the analysis nor the correctness of the algorithms requires us to lay out the tree in any particular way. However, as we saw in Section 3.1.3, page 32, in case of binary search and as shown through experimental analysis in [BFJ02] and [LFN02], a well-chosen layout of the tree can yield a significant increase in performance.

As we have seen in Section 3.1.3, using the van Emde Boas layout for binary search trees gives an asymptotical reduction in the number of memory transfers incurred. That is not the case in when dealing with funnels. However, as argued in the proof of Theorem 4-3, fill does a number of tree operations, including recursive call invocations, flip operations, etc., proportional to the total number of comparisons and moves

[1] Our implementation does not current exploit this observation. It allocates a full tree, but never actually visits the leaf nodes.

performed, and thus visits a new node a significant number of times. Laying out nodes, so they are near each other, may then increase the algorithms overall locality significantly.

Aside from increasing locality, using controlled layout allows us to compute the position of the nodes relative to each other. This eliminates the need for accessing pointers to children stored in nodes and the potential data hazard in the pipeline. This latter aspect may well be as important as the first.

Implementation

In the implementation of the funnel, we use the STL concept of an allocator. The allocator is simply a class, through which algorithms can dynamically create objects. All containers in the STL provide a way for the user to supply an allocator. By abstracting away the allocation mechanism, the user of the containers is free to provide their own allocators and thereby control how objects are dynamically created. Such a mechanism is also provided through the new operator; however, the new operator is global and cannot be customized on a per algorithm or per container basis. If the user does not supply an allocator a default allocator, std::allocator, is used. std::allocator in turn uses the new operator.

The construction and destruction of funnels are done through a layout class template. Its interface is simple; the only reason for putting this functionality in a class is that we may parameterize funnels over different implementations.

```
template<class Navigator, class Splitter, class T, class Allocator>
class layout
{
public:
    typedef typename Navigator::node node;
    static node *do_layout(int order, Allocator& alloc);
    static void destroy(node *root, int order, Allocator& alloc);
};
```

The interface consists of two static member functions: do_layout and destroy. do_layout allocates and lays out a complete tree of a given order and returns a pointer to the root, and destroy tears down and deallocates the tree. It is parameterized by a navigator (see below), a splitter defining the size of the buffers, the type of elements in buffers and finally the allocator.

The Splitter plays the important role of deciding at what height we split the funnel when doing the van Emde Boas recursion (our implementation simply returns $\lfloor h/2 \rfloor$, with h being the height of the funnel being split) and what capacity the output buffer of a k-funnel should have. As such, it is used extensively throughout the implementation of both the funnel and funnelsort.

A given layout implementation ensures that allocating nodes and arrays of elements is done in a specific order. Achieving correct layout then relies on the allocator fulfilling memory requests in a contiguous manner. Our implementation includes an allocator, stack_allocator that does this by allocating a large chunk of memory once using new and move and return a pointer into this chunk, in response to memory requests. The amount of space needed for the initial allocation has to be determined at allocator construction time. For general mergers, the space needed is computed exactly by the static size_of member function.

Mixed and Pooled Layouts
Each layout comes in two variants. One that, as described below, allocates nodes and buffers intermixed, and one that allocates a pool of elements in which the buffers are placed. The first variant is called mixed, the latter pooled. The pooled layouts result in nodes being allocated together, much like the search trees of [BFJ02] followed by buffers laid out contiguously.

The van Emde Boas Layout
We have already discussed the van Emde Boas layout. In our implementation, it is realized by first recursively laying out the top tree then for each bottom tree from left to right, allocating its output buffer then recursively laying out the tree. Figure 5-2 shows a funnel laid out in the array below it. Note the `stack_allocator` allocates backwards.

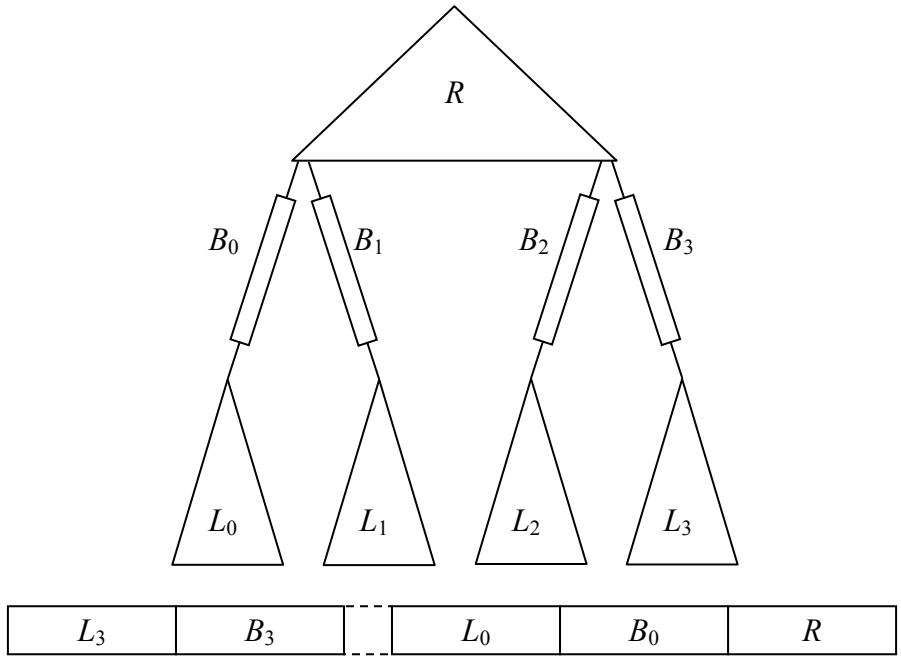


Figure 5-2. The van Emde Boas mixed layout.

Breadth-first Layout
The breadth first layout was the layout used in [Wil64] for implementing heaps. The nodes of the tree are allocated in the order they are visited by a left-to-right breadth first traversal of the tree. This is achieved by recursively allocating a funnel of height one smaller and then allocate the leaf nodes and their output buffers from left to right. The number by the nodes and buffers in Figure 5-4 show the order in which they are allocated. The numbers in Figure 5-3 shows the relative positions of nodes and buffers, when using pooled breadth-first layout.

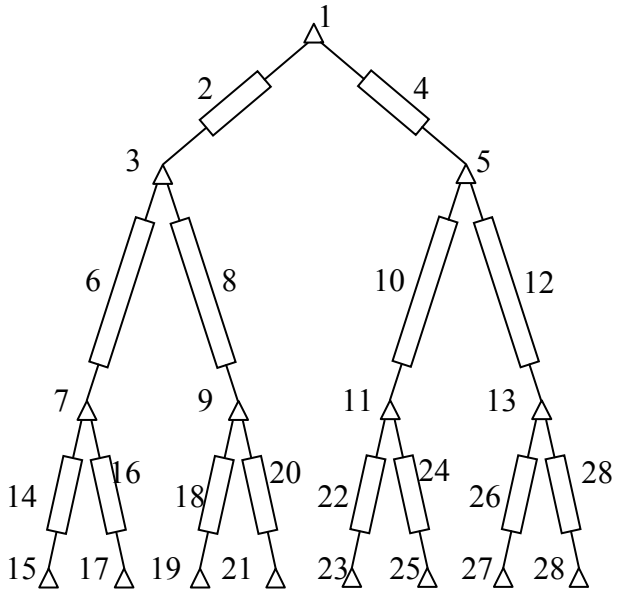


Figure 5-3. Breadth-first pooled layout.

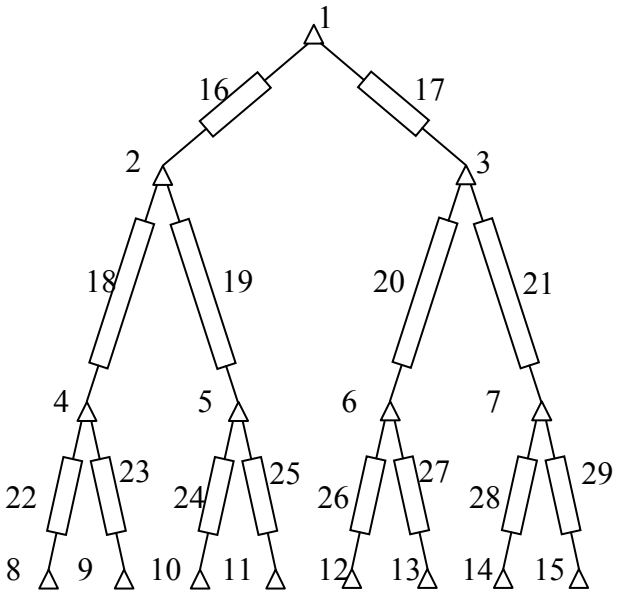


Figure 5-4. Breadth-first mixed layout.

Depth-first Layout

In the depth-first layout, the nodes and their output buffers are allocated in the order they are visited in a left-to-right depth first traversal of the tree. This is achieved by allocating the root and recursively allocate the subtrees below it from left to right. Before allocating the subtrees, their output buffer is allocated. The order can be seen in Figure 5-5.

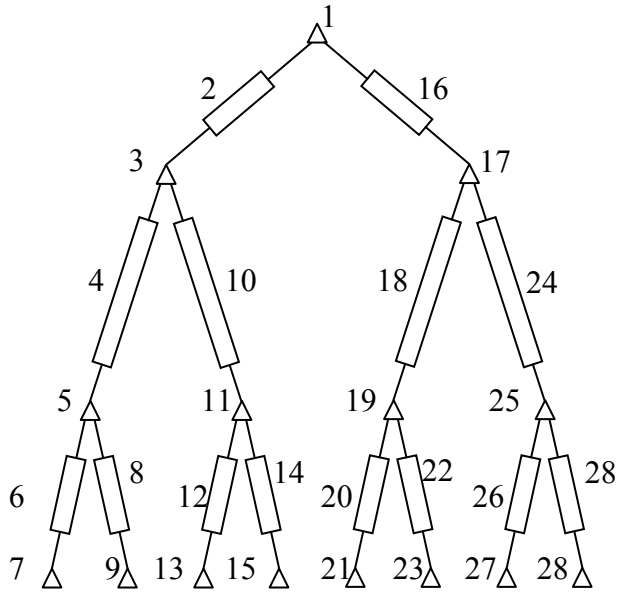


Figure 5-5. Depth-first mixed layout.

5.3.3 Navigation

A class known as a *navigator* is responsible for locating the parts of the funnel. Confining this functionality to a class allows us to experiment with different ways of traversing the funnel. Its interface is as follows:

```
template<class Node, class Splitter>
class navigator
{
public:
    typedef ... token;
    typedef ... bookmark;
    typedef Node node;
    typedef typename Node::stream buffer;
    token parent();
    token child(int i);
    navigator& operator+=(token t);
    navigator& next_dfs();
    template<class Functor>
    Functor enum_buffers(Functor f);
    level_iterator begin_level(int depth);
    level_iterator end_level(int depth)
    bookmark mark() const;
    bool operator==(bookmark m) const;
    bool operator!=(bookmark m) const;
    bool is_root() const;
    bool is_leaf() const;
    buffer *input();
    buffer *output();
};
```

Navigators resemble iterators in that they represent a single node in a data structure; however, they are capable of going in more directions than forward and backward. To

support this in a generic way, we introduce a new token type, used to represent directions. Going to the parent is one direction and going to each of the children is another. The navigator is responsible for flipping buffers it passes.

The begin_level and end_level member functions provide a way to iterate through nodes on a specific level. level_iterators are bidirectional iterators dereferencing to pointers to nodes. This is used to tell the nodes where their children are placed during the construction and layout of the tree (hence the dependency of layout class on navigator classes). next_dfs moves the navigator to the next node in a search, where the nodes are enumerated in a way that when we visit a node, we have visited all nodes below it. This is used in the warm-up phase. enum_buffers provides for a way of enumerating buffers. This is used for resetting the merger for and emptying the buffers.

A simple implementation of a navigator is one that relies on the nodes to supply the address of their children and parents, but that requires the space in each node and the navigator to access these pointers. We define four categories of nodes, based on the information stored in them:

- *Simple node*. A node that stores nothing but the head and tail of its input streams.
- *Flip node*. A simple node that also stores the beginning and end of their input buffers. As the name implies, these nodes can flip their own input buffers.
- *Pointer node*. A simple node also storing the address of its children.
- *Pointer flip node*. A flip node also storing the address of its children.

Navigators that are more sophisticated will require less information of the nodes. Note that no category of nodes requires the node to store pointers to its parents. The reason for this is that the navigator can store pointers to nodes on the path to the current node, on a stack using much less space. On another stack, navigators keep information about output buffers on the path from the root to the current node. This is to avoid accessing data in the parent node.

A general pointer_flip_navigator has been implemented. It requires the funnel to be built using pointer flip nodes and all operations are implemented using the two mentioned stacks and the information stored in the nodes. Aside from the two stacks, a pointer to the current node is maintained and tokens are simply pointers to nodes.

If we choose to use the default allocator, we have no guarantee of where nodes and buffers are placed, so we are forced to use pointer flip nodes and the pointer_flip_navigator. When using stack_allocator and the mixed variants of the layouts, we know that the output buffer of a node lies immediately after the node itself. Using pointer nodes, the beginning of the i'th buffer can be obtained by adding the size of a node to the address of the i'th child, thus providing the information needed in a flip operation. When using the stack_allocator, we know where the nodes and buffers are placed. Our implementation includes navigators that exploit this for all pooled layouts and for the mixed variant of the van Emde Boas layout. For pooled layouts, we must use at least flip nodes, while the navigator for mixed van Emde Boas only requires simple nodes. We compute the address of the parents and children of each layout in the following way:

For the pooled breadth-first layout, the i'th child (counting from 0) of a node positioned at index j is located at index $(j\text{-}1)z\text{+}i\text{+}2$ and its parent is located at index $\lfloor(j\text{-}2)/z\text{+}1\rfloor$. For the pooled depth-first layout, we use a measure d that is the distance between the child nodes. When at the root, d is the number of nodes in a full tree of

height one smaller than the funnel. The i'th child of a node positioned at index j is then located at address $j+id+1$. When going to a child we integer divide d with z. When going to the parent, we multiply and add one, and the index becomes $j-id-1$, where the node is the i'th child of the parent. i is computed as $(k+z-2) \bmod z$, with k being the breadth-first index. The result of these operations gives us the index of a node in the layout, with the root located at index 1. The final address is then computed by subtracting this index from the known location of the root. For this to work, we use perfect balanced trees as discussed below.

For the mixed van Emde Boas layout, we observe that when following the recursion until the node is the root of a bottom tree, it will be at an offset from the root of the top tree given by the size of the bottom trees and their output buffers times the number of bottom trees to the left of the child plus the size of the top tree. The number of bottom trees to the left is $k \bmod (n+1)$, with n being the number of nodes in the top tree and k being a breadth-first-like index that is updated with $kz+i$, when going to the i'th child and $\lfloor k/z \rfloor$ when going to a parent. The size of the bottom tree is kept is in a pre-computed table B as is the depth D of the root of the top and the number of nodes in the top tree N. These tables can be computed with one entry per level of the tree, since the recursion unfolds the same way for all nodes on the same level. The address of the nodes on the path from the root to the current node is kept in a table P, so address of the root of the top tree is $P[D[d]]$ with d being the depth of the current node. The last ingredients is a table T with the size of the top tree. The address of the i'th child then becomes

$$P[d] = P[D[d]] - \Big(\big(k \bmod \big(N[d]+1\big)\big) B[d] + T[d]\Big) \tag{5.1}$$

We know that $N[d] = (z^j-1)/(z-1)$ for some integer j. With $z = 2$, we can thus compute $k \bmod (N[d]+1)$ as $k \text{ and } N[d]$ which is likely to be faster. With the parameter z an integer template argument, we select the faster way through partial template specialization. For pooled layout, the modification lies in that buffer sizes should not be included in the offset from the root of the top tree. This in turn makes the table T identical to table N, so one of them can be discarded. For general z and $z = 2$ respectively, we get

$$P[d] = P[D[d]] - \Big(\big(k \bmod \big(T[d]+1\big)\big) B[d] + T[d]\Big) \tag{5.2}$$

$$P[d] = P[D[d]] - \Big(\big(k \text{ and } T[d]\big) B[d] + T[d]\Big) \tag{5.3}$$

The relation in (5.3) is in turn what was used in [BFJ02] for navigating optimal cache-oblivious binary search trees.

All implementations of navigators require storing multiple tables. This will make it, unlike iterators, infeasible to pass them as arguments to functions and in turn, unsuitable for use in recursive functions. Our implementation of fill is thus based on an unfolded recursion. Using the navigator abstraction and an unfolded recursion may increase the overall instruction count beyond what is possible using the simple recursive scheme of Algorithm 4-4. To clarify this, a recursive implementation was also made, however it requires the use of pointer flip nodes.

Non-power-of-z-funnels

Before proceeding with the experiments with layout and navigation, a point about unbalanced trees need to be made. The problem is that if we want to merge z^h+1 streams, we may have to lay out a z^{h+1}-funnel, which takes up a lot more space. It is indeed not necessary to lay out the entire z^{h+1}-funnel, since some of it will not be used. Figure 5-6 shows how to conserve space by not constructing an entire funnel.

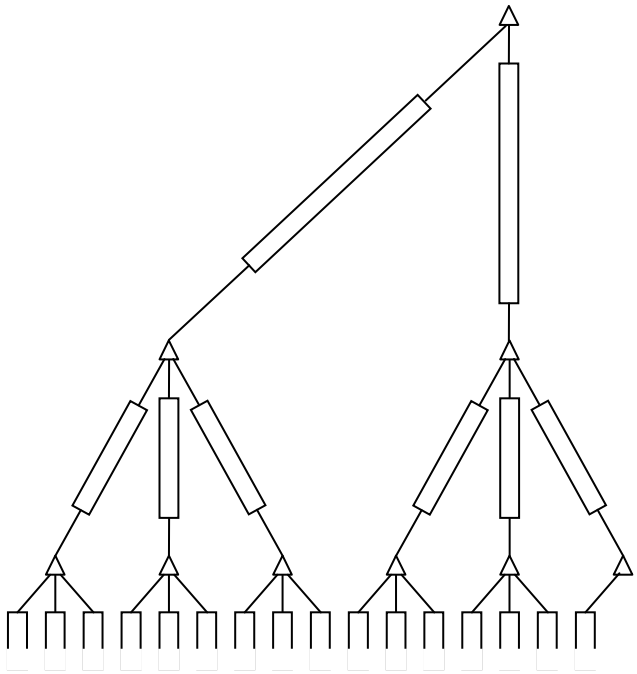


Figure 5-6. A merge tree of order 16 and $z = 3$.

However, not laying out a complete tree will foil all of the implicit navigation schemes described above, thus layout classes are asked to layout balanced trees when using implicit navigators. When using pooled layout, only the nodes of the balanced tree are laid out; the buffers are not. For the mixed van Emde Boas layout, however, we need to lay out a fully balanced funnel. When z becomes large, this may matter for certain values of k.

Test Results

For constructing funnels, we now have the choice between three layouts with two variants each, the choice of using the default allocator or using the stack_allocator, and the choice of using an implicit navigator or a general pointer_navigator or pointer_flip_navigator. This amounts to a total of $3{\cdot}2^3 = 24$ combinations, however, we cannot use implicit navigators with the default allocator, so a fourth of the combinations cannot be used. Furthermore, implicit navigators for mixed depth-first and mixed breadth-first have not been implemented. A total of $¾{\cdot}24\text{-}2 = 16$ combinations remain. They are listed in Table 5-1.

Name	Layout	Allocator	Node	Navigator
pb_heap_mveb	Mixed van Emde Boas	std::allocator	Pointer flip	Pointer flip
pb_stack_mveb	Mixed van Emde Boas	stack_allocator	Pointer	Pointer
impl_mveb	Mixed van Emde Boas	stack_allocator	Simple	Implicit
pb_heap_veb	Pooled van Emde Boas	std::allocator	Pointer flip	Pointer flip
pb_stack_veb	Pooled van Emde Boas	stack_allocator	Pointer	Pointer
impl_veb	Pooled van Emde Boas	stack_allocator	Flip	Implicit
pb_heap_mbf	Mixed breadth-first	std::allocator	Pointer flip	Pointer flip
pb_stack_mbf	Mixed breadth-first	stack_allocator	Pointer	Pointer
pb_heap_bf	Pooled breadth-first	std::allocator	Pointer flip	Pointer flip
pb_stack_bf	Pooled breadth-first	stack_allocator	Pointer	Pointer
impl_bf	Pooled breadth-first	stack_allocator	Flip	Implicit
pb_heap_mdf	Mixed depth-first	std::allocator	Pointer flip	Pointer flip
pb_stack_mdf	Mixed depth-first	stack_allocator	Pointer	Pointer
pb_heap_df	Pooled depth-first	std::allocator	Pointer flip	Pointer flip
pb_stack_mdf	Pooled depth-first	stack_allocator	Pointer	Pointer
impl_df	Pooled depth-first	stack_allocator	Flip	Implicit

Table 5-1. The possible combinations of layout and navigation available in our implementation

The 12 non-implicit combinations are also implemented with pointer flip nodes using a pure recursive fill function. Their names have "pb" exchanged with "rec".

For the experiment, we have had each of the 24 funnels merge k streams of k^2 elements each using k-funnels with $z = 2$, $\alpha = 1$ and $d = 2$ and $k = 15, 25, \ldots 270$. The streams are formed by allocating an array of k^3 pseudorandom elements (pairs of long and void*) and sorting sections of size k^2 with std::sort. The funnel is constructed the streams attached, elements merged, and the merger reset $\lfloor 20{,}000{,}000/k^3 \rfloor$ times. The time measured is the time it takes to do this, save for the construction of the funnel. The output of the funnel is not stored anywhere; it is simply passed through an output iterator that checks whether the elements are sorted.

To avoid having to display 24 data series in the same chart, the result of the experiment is presented as a tournament with a group of implicits, a group of pointer navigators using default allocator, one using stack_allocator, one recursive using default allocator, and finally one using stack_allocator. From each group we choose a winner to appear in the final chart. The result is as follows. First, let us look at implicit navigation.

5.3.3 Navigation

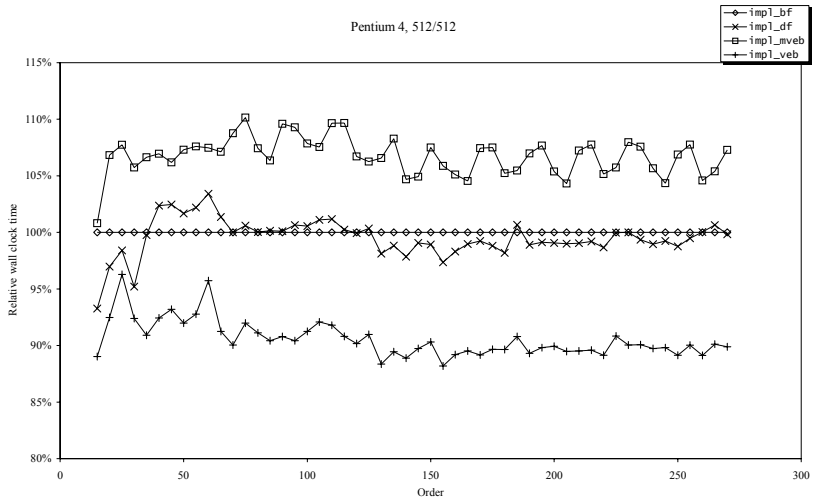


Chart C-1. Implicit layout on Pentium 4.

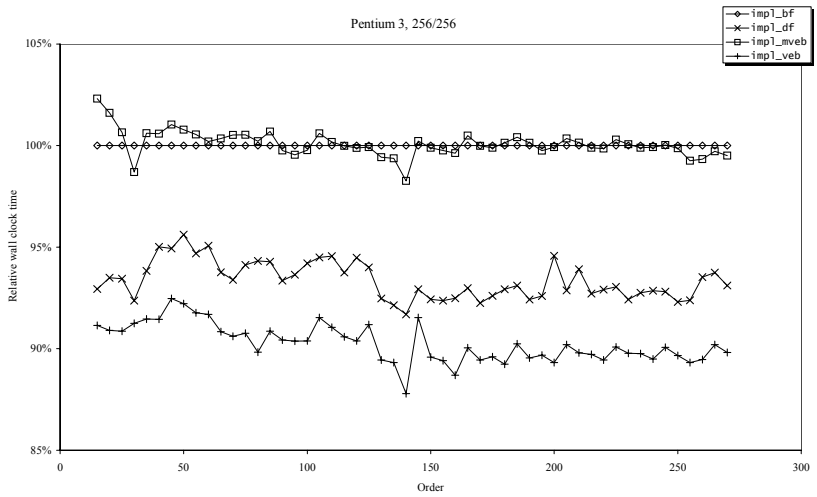


Chart C-2. Implicit layout on Pentium 3.

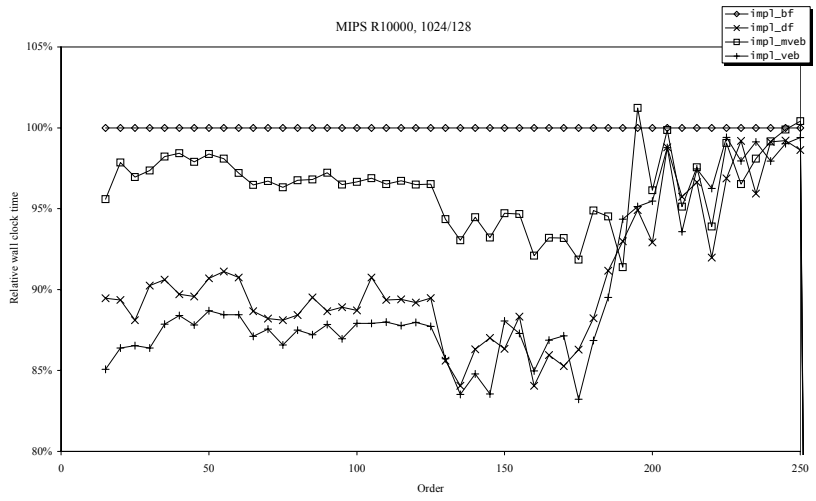


Chart C-3. Implicit layout on MIPS 10000.

The charts are normalized to breadth-first layout, which on both the Pentium 3 and the MIPS architectures are clearly the worst performers, even though it has the smallest instruction count. The reason for this must be effects in the memory system. The measurements made by PAPI on the MIPS, indicates that it is not as much the L2 cache rather the TLB that makes the difference:

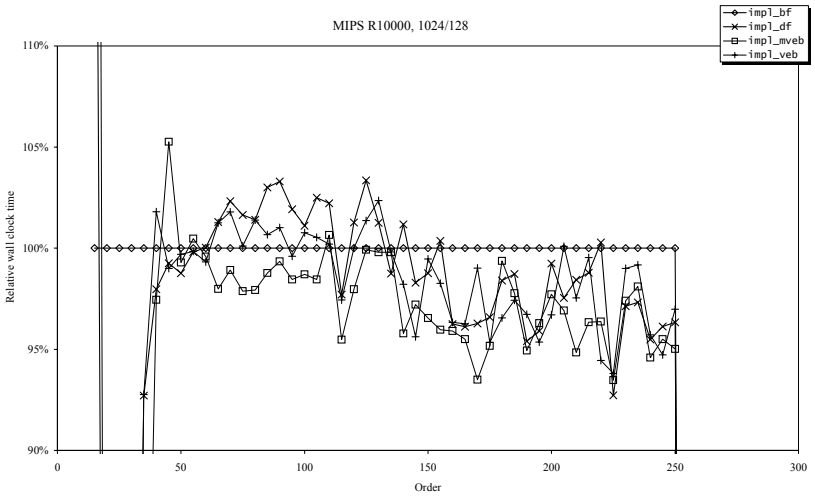


Chart C-4. Implicit layout on MIPS 10000, relative L2 cache misses.

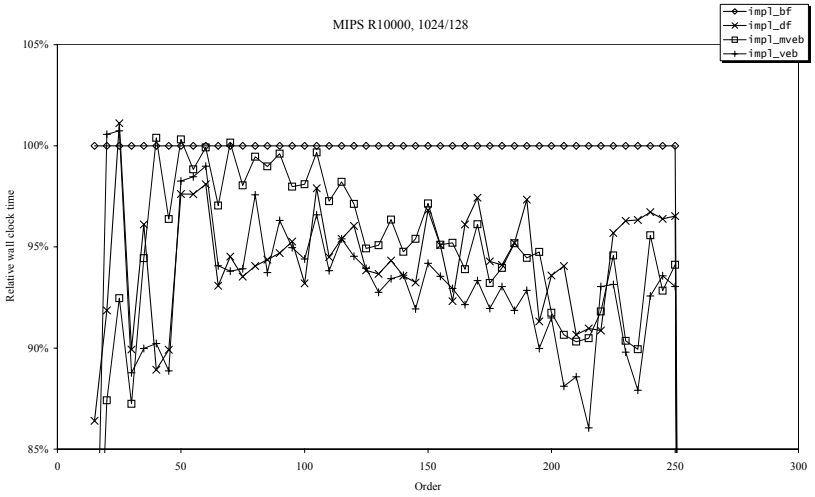


Chart C-5. Implicit layout on MIPS 10000, relative TLB misses.

The L2 cache incurs about the same number of misses regardless of layout, perhaps with the breadth-first incurring more misses; however, for large funnels (height at least six), we can see that some layouts are more "TLB friendly" than others are. TLB misses are handled in software on the MIPS so the performance penalty is greater. The breadth-first layout exhibits least locality as observed in [BFJ02]. The reason for this is that the parent is, except at the top, always located far from its children. All but the left most child are also placed far from the parent in depth-first layouts as well; however, for $z = 2$, that is half the children of a node, so it is not such a significant effect.

Navigating a pooled depth-first funnel requires us to update three variables when going from node to node. All three architectures are super-scalar, so these updates can occur in parallel and need thus not take any longer than just updating the breadth-first index.

That the mixed van Emde Boas layout does not suffer from being forced to lay out balanced trees is also noteworthy. The fact that the buffers are not touched during the fill phase of the merge contributes to this. The best combination seems to be the pooled van Emde Boas Layout on all architectures. It has good locality and the navigation is not as complex as the mixed van Emde Boas layout, which may have even higher locality. Hence, we choose the pooled van Emde Boas Layout as the winner of this group.

Let us now turn to the pointer-based navigators. When using the default allocator, none of the architectures seems to prefer any of the layouts particularly. As an example, the result from the Pentium 3 can be seen here:

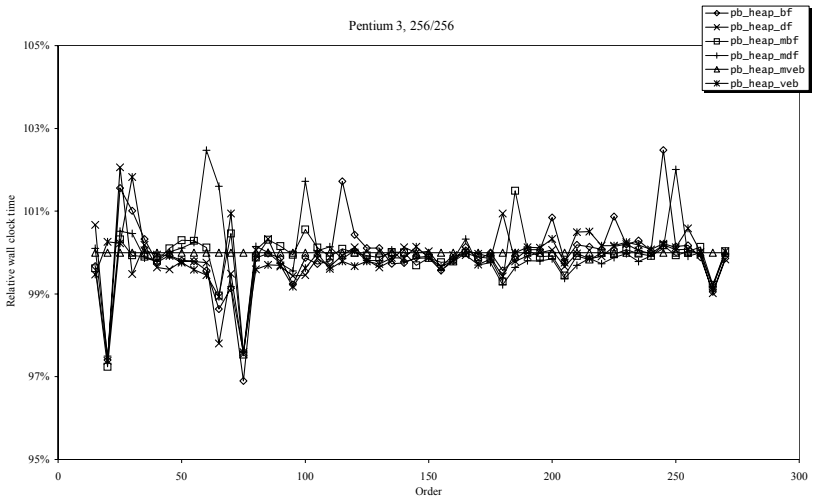


Chart C-7. Layout using std::allocator on Pentium 3.

We choose the depth-first layout for the final, because the Pentium 4 shows a slight (<1%) shift in its favor. The picture changes when using the stack_allocator:

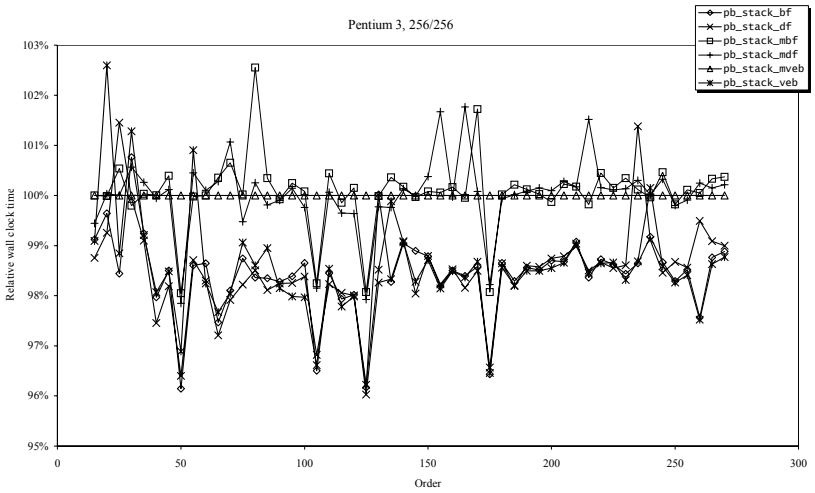


Chart C-12. Layout using stack_allocator on Pentium 3.

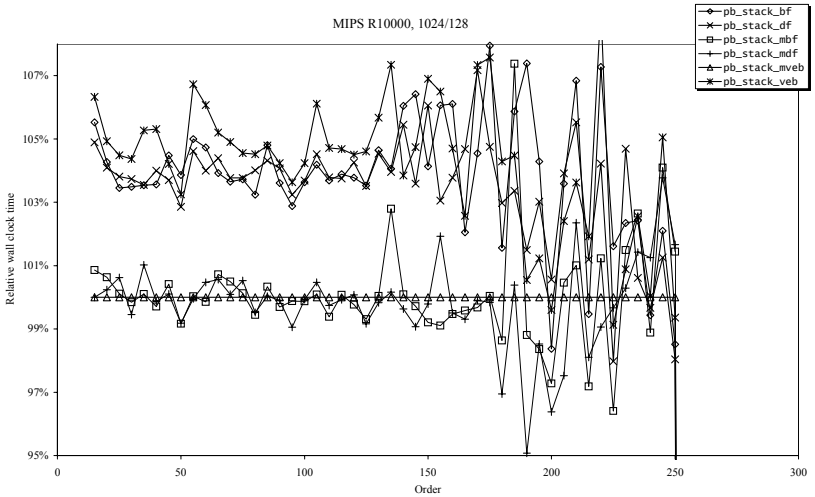


Chart C-13. Layout using stack_allocator on MIPS 10000.

The Pentium 3 (as well as the Pentium 4) clearly favors the pooled layouts, while the MIPS favors the mixed. There does not seem to be any special preference in the PAPI results. One explanation could lie in the lower level caches; when using pooled layouts, all the nodes can fit in L1 cache and if the associativity of the L1 cache is sufficiently high, they will likely stay there. However, if the cache has low associativity, it is likely that it cache lines with nodes on them will be evicted due to conflict misses during operations elsewhere in the funnel. In the latter case, it is probably best to store the nodes near the action. As it is, the Pentium 4 has a four-way set associative L1 cache while that of the MIPS is only two-way set associative. We choose in favor of the Pentiums and send the pooled depth-first layout to the final.

When using the recursive implementation of fill, the Pentium 4 again has no preferences with less than 2% difference in performance. The Pentium 3, however, seem to prefer the pooled layouts not only with the stack_allocator, but also when using the default allocator:

5.3.3 Navigation

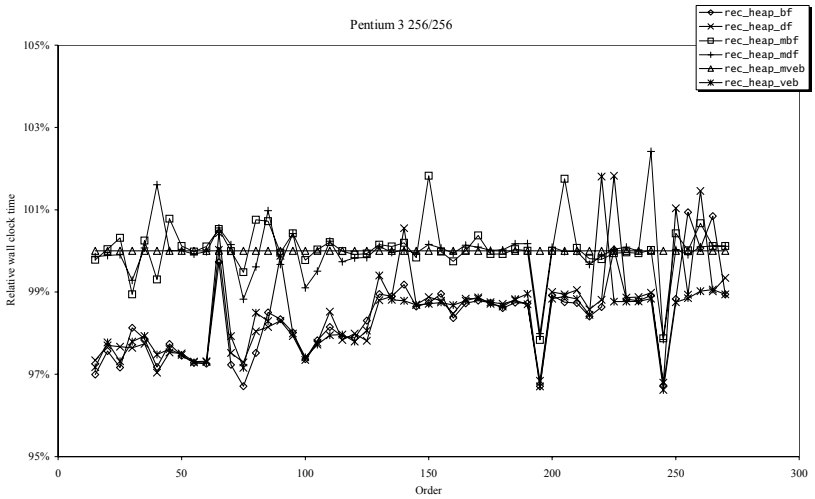


Chart C-17. Recursive fill, heap, Pentium 3.

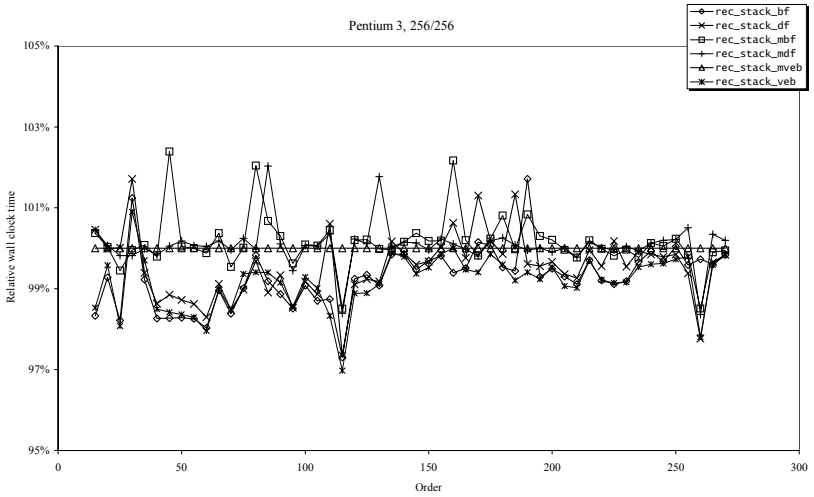


Chart C-22. Recursive fill, stack, Pentium 3.

The MIPS fortunately seem to have lost its interest in the mixed layouts:

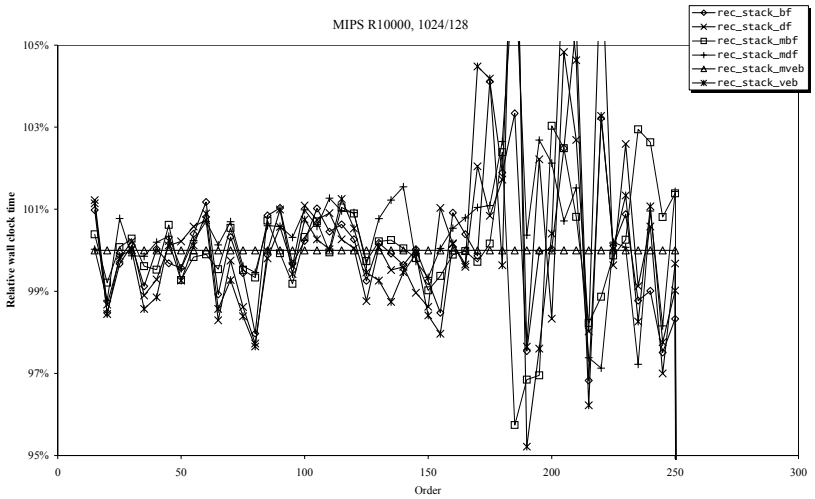


Chart C-23. Recursive fill, stack, MIPS 10000.

We choose the pooled van Emde Boas layouts for the final. Now that we have chosen a good layout from each of the groups, it is time to compare them to each other, now normalized to stack based layout with recursive navigation implementation.

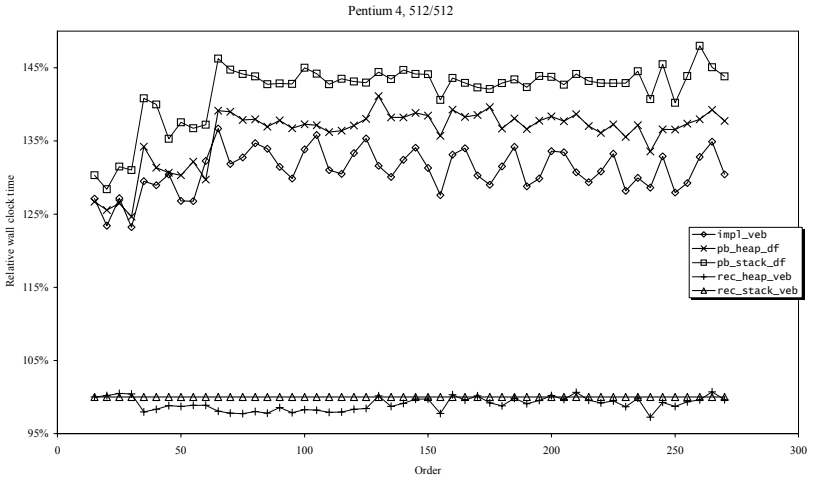


Chart C-26. Final, Pentium 4.

5.3.3 Navigation

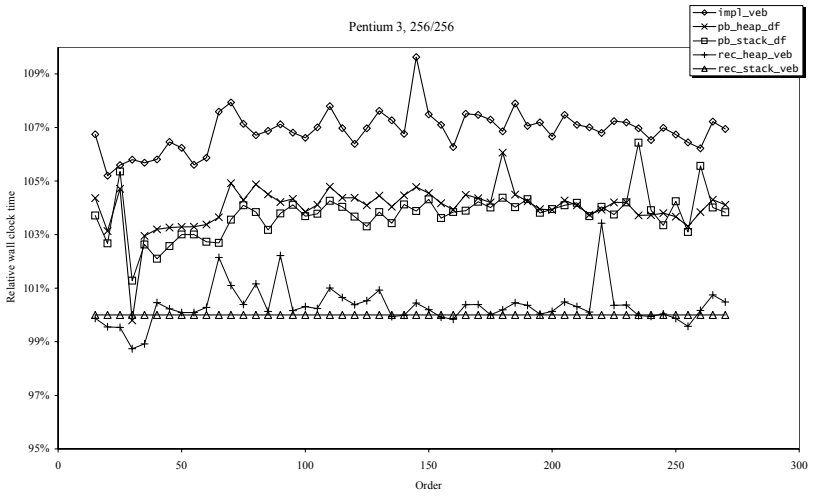


Chart C-27. Final, Pentium 3.

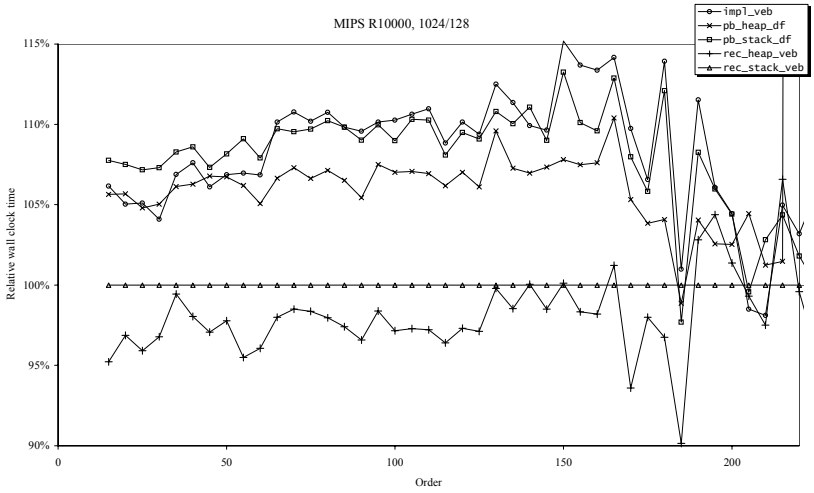


Chart C-28. Final, MIPS 10000.

We see that the Pentium 4 gains tremendously from using a recursive implementation of fill. This can be contributed to the number of transistors dedicated to avoiding control hazards. The Pentium 4 has a special return address stack, used by the fetch unit when returning from a function call. The stack contains the address of the next instruction to be fetched, which will then be ready immediately. When recursions are not too deep (as is the case here), this approach is far better than using conditional branches in the loops of the unrolled recursion. The effect is far from as pronounced on the Pentium 3 and the MIPS, where the effect is more likely due to overall lower instruction count.

We can also see that the implicit navigation is competitive only when on equal terms, comparing to the pointer-based navigators. When comparing implicit with the recursive algorithm, the simple recursive approach performs much better. Moreover, it turns out that whether using controlled layout through stack_allocator or leaving it to the heap allocator does not make a significant difference. Indeed, MIPS tend to favor

memory delivered directly by the heap allocator. A reason for this is that the heap allocator is system specific and thus has detailed knowledge of the system parameters. This in turn allows it to allocate memory that is e.g. aligned on cache line boundaries.

Conclusion
In all, we conclude that the effects of the layout, and in turn the effects of cache, are dwarfed by other aspects. The key to achieving high performance in funnel implementations is through simplicity, rather than complex layouts. However, a good layout, such as depth-first or the van Emde Boas, seems to give a couple of percent on the performance scale.

5.3.4 Basic Mergers

By far the most time in a good funnel implementation should be spend merging elements. In our implementation, this means the basic mergers. Making sure they are performing optimally is thus important to achieving overall high performance.

The body of the fill algorithm (page 48) essentially implements the basic_merger application operator. When calling add_stream on a basic_merger, if the stream is not empty a counter, named active, is incremented and the stream and the associated token is stored. If it is empty, it is simply ignored. Upon invocation, the basic merger will check active. If it is zero, it returns immediately. If it is one, the contents of the only stream are copied to the output. If the input got empty, we return its token; otherwise, we return the output token. If active is greater than one, the actual merging begins. The implementation of the merging is put in a member function named invoke.

Binary Mergers
There are a couple of subtleties concerning the use of basic mergers, which we will now discuss. The streams added to the basic merger is a part of the object state and are as such accessed through the this pointer. In general, this will cause a slight overhead every time we access them, which is a couple per element merged. However, basic_mergers are stack objects of the fill function, so in fill, the this pointer is a compiletime computable constant offset from the stack pointer. Provided the operator() is inlined into the fill function, this will also merely be a constant offset from the stack pointer, thus the member variables will act as if they are normal stack variables and can as such be accessed without having to dereference the this pointer. Nonetheless, even though we insist that the compiler should inline the functions, the speed is increased if we make local copies of the member variables. This must be contributed to poor code generation on behalf of the compiler.

Another subtle issue that cannot be attributed to the compiler is the aliasing problem, that arises from passing the begin iterator of the output by reference. In such situations, the compiler cannot in general be certain that the iterator (which is often just a pointer) does not reference another iterator, in particular one of the input iterators. This in turn means it has to generate code that updates the referenced iterator and not just a local copy, each time the output iterator is updated. This turns writing elements to the output into a double dereferencing and incrementing the output begin iterator a load-increment-store instead of just an increment.

Since we know the output begin pointer is unique, and a reference to it does not reference any other pointer, we can solve this problem by explicitly making a local

copy of it, use that for merging, and write it to the referenced iterator before returning. The complete merging code implementing Algorithm 4-4 looks like this:

```
template<class FwIt, class T, class Comp>
inline Token invoke(FwIt& b, FwIt e, Token outtoken, Comp& comp)
{
    typename Stream::pointer head[2] =
        { stream[0]->begin(), stream[1]->begin() };
    typename Stream::pointer tail[2] =
        { stream[0]->end(), stream[1]->end() };
    FwIt p = b;
    while( p != e )
    {
        if( comp(*head[0],*head[1]) )
        {
            *p = *head[0], ++head[0], ++p;
            if( head[0] == tail[0] )
            {
                outtoken = token[0];
                break;
            }
        }
        else
        {
            *p = *head[1], ++head[1], ++p;
            if( head[1] == tail[1] )
            {
                outtoken = token[1];
                break;
            }
        }
    }
    *stream[0] = Stream(head[0],stream[0]->end());
    *stream[1] = Stream(head[1],stream[1]->end());
    b = p;
    return outtoken;
}
```

This basic merger implementation is called simple_merger. We see that each time a single element is merged in the funnel at least three conditional branches have to be evaluated, namely the branch in the while loop, the branch on which head is smaller, and the branch on whether the input got empty. This could be a major overhead. However, due to sophisticated branch prediction techniques, predictable branches need not cause any performance penalty. The test that branches on which head element is smaller is inherently unpredictable; however, we expect the loop branch and the branch on empty input to be more predictable.

Consider a funnel with height power-of-two. No rounding is necessary when following the van Emde Boas recursion, so between every other level, there is a buffer of size αz^d. With $\alpha = 1$, $z = 2$, and $d = 3$, these buffers can contain eight elements. This means that at most eight elements can be merged before one of the two branches something different from the last time and cause a pipeline flush. This is not a lot. A quick fix would be to increase α, but this will not make the per merged element branches go away. We could also look at the problem more intelligently; since we know these branches will not fail (in the sense that they cause the loop to break) until

enough elements have been moved to either make the output full or on of the inputs empty. We can see that the number of elements will be at least the minimum of the number of elements in the input streams and the space available in the output. The adapted loop then looks like this:

```
Diff min = e-p;
if( tail[0]-head[0] < min )
    min = tail[0]-head[0];
if( tail[1]-head[1] < min )
    min = tail[1]-head[1];
do
{
    assert( min );
    for( ; min; --min )
        if( comp(*head[0],*head[1]) )
            *p = *head[0], ++head[0], ++p;
        else
            *p = *head[1], ++head[1], ++p;
    min = e-p;
    if( tail[0]-head[0] < min )
        min = tail[0]-head[0];
    if( tail[1]-head[1] < min )
        min = tail[1]-head[1];
}
while( min );
```

which we denote the two_merger. The benefit of this approach is that we have eliminated one of the branches from the core merge loop, but at the price of having to compute the minimum now and again. However, the minimum can be computed entirely without using branches, namely by using conditional move instructions, so the overhead should be small. A worst-case scenario would be an input buffer consisting of a single large element, the other input of many small elements, and plenty of space in the output. The single element would cause the minimum to be one and thus the minimum to be recomputed every time one of the small elements is moved to the output.

To get a feel for how often such asymmetrical stream sizes occur, we counted the number of times the smallest input stream was a given fraction of the size of the largest stream. The resulting distribution can be seen in Figure 5-7. This was obtained through a full run of funnelsort on 0.7 million, 7 million, and 16.3 million uniformly distributed elements with $\alpha = 16$ and $d = 2.5$.

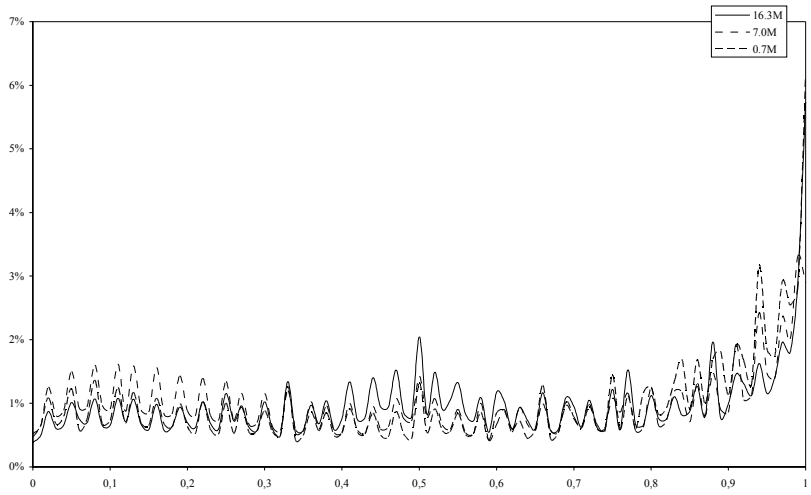


Figure 5-7. The distribution of relative sizes of input streams of basic mergers.

We can see that there is a slight tendency for the input streams to be of equal size; however, in general the small input stream can have a size any fraction of the size of the large input stream. Thus, we do not expect extremely small streams to be merged with very large streams with any significant frequency.

Still, perhaps we can gain further performance if we used a merge function that took into account the fact that sometimes we need to merge smaller streams with large streams and do that more efficiently. [Knu98] includes a description of an algorithm (Algorithm H, Section 5.3.2) originally due to F. K. Hwang and S. Lin that achieves near-optimal number of comparisons on inputs of this type. The adaptation of it to the basic merger setting is slightly tricky so we leave it out and refer to the accompanying source, where it is implemented as the `hl_merger`. It has a significant overhead but it may be that it is outweighed by the frequency of asymmetrical stream sizes. From a theoretical perspective, this merger can decrease the total number of comparisons performed in the funnel.

Realizing that the overhead of these more clever mergers may hamper their performance, we could also employ hybrid mergers; mergers that only use clever tricks under certain conditions. The `hyb3` merger checks the relative size of the input streams. If the size of one stream is more than four times the size of the other, the `hl_merger` is used. Otherwise, the `two_merger` is used, but only as long as minimum is at least eight. From then on, it uses `simple_merger`. The `hyb` merger is a hybrid of only `two_merger` and `simple_merger` also with a cutoff at minimum of eight. The `hyb0` only computes minimum once does one iteration of `two_merger` and proceeds with `simple_merger`. The reason this makes sense is that about half the times a basic merger is invoked, the minimum will be determined by the space available in the output, since on every other level of the funnel, the output buffer has a larger capacity than the input buffers. If that is the case, the minimum computed will be the exact number of elements moved during the entire merge. If it is not the case, we continue with `simple_merger` to minimize overhead.

We performed the same benchmark as with the analysis of layout and navigation. Here we used the `rec_heap_mveb` and realizing that the choice of constants α and d can

be significant we ran the test for $(\alpha,d) = (1.0, 3.0)$, $(4.0, 2.5)$, and $(16.0, 1.5)$. With these parameters, the smallest buffers are of size 8, 23, and 45, respectively. The results for $(\alpha,d) = (1.0, 3.0)$ can be seen here:

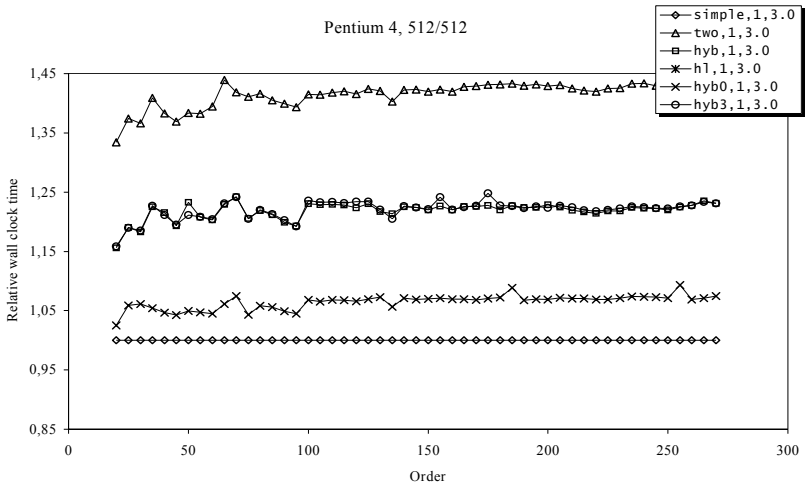


Chart C-31. Basic mergers, $(\alpha,d) = (1,3)$, Pentium 4.

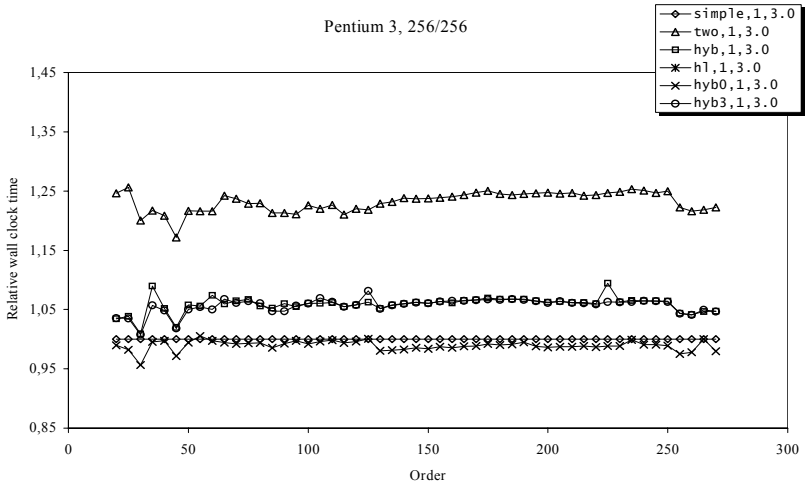


Chart C-32. Basic mergers, $(\alpha,d) = (1,3)$, Pentium 3.

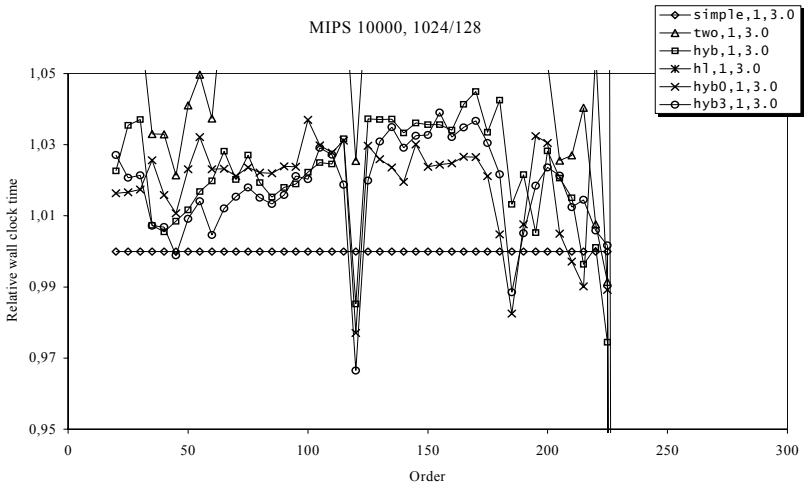


Chart C-28. Basic mergers, $(\alpha,d) = (1,3)$, MIPS 10000.

The MIPS produces a lot of noise (note the scale); however, it is clear that with a minimum buffer size of eight, not enough elements are merged per basic merger invocation to warrant the use of any method that has an overhead associated with it on any of the architectures. The overhead of the hl_merger made the entire merge take at least three times longer and is thus far off scale. In addition, the difference in all benchmarks between the hyb_merger and the hyb3_merger is minimal, implying that cases where the smaller stream has less than a forth the number of elements of the large stream are rare and that in those cases using the hl_merger neither improves nor worsens the performance.

Going from $(\alpha,d) = (1.0, 3.0)$ to $(4.0, 2.5)$ the two_merger does not gain much, but the hybrids start to get competitive at least on Pentium 3 and MIPS. Going to $(16.0, 1.5)$, the Pentium 4 finally seems to benefit from the tighter inner loops of the hyb0_merger; however, using the pure two_merger still incurs a 35% running time increase.

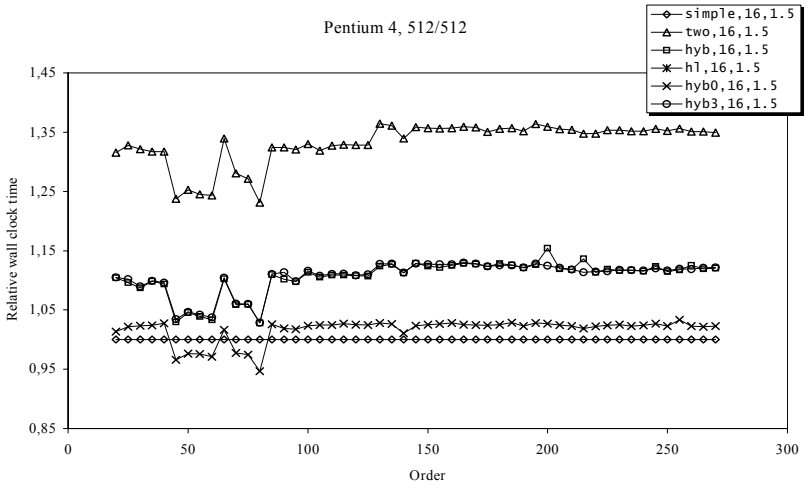


Chart C-37. Basic mergers, $(\alpha,d) = (16,1.5)$, Pentium 4.

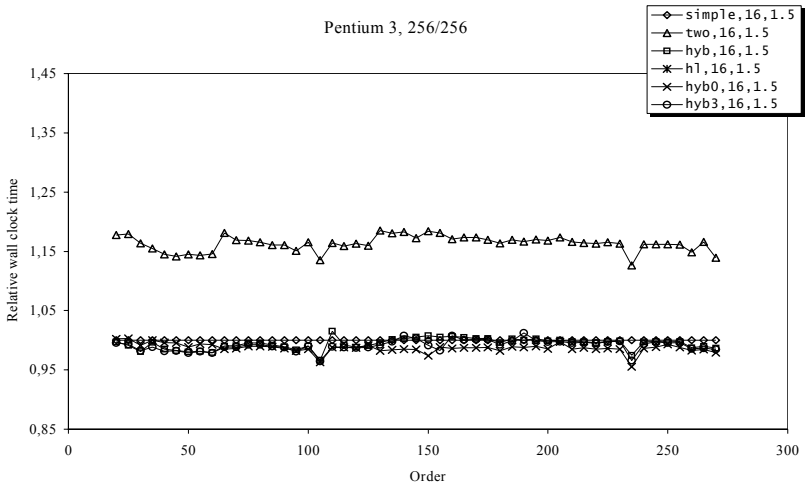


Chart C-38. Basic mergers, $(\alpha,d) = (16,1.5)$, Pentium 3.

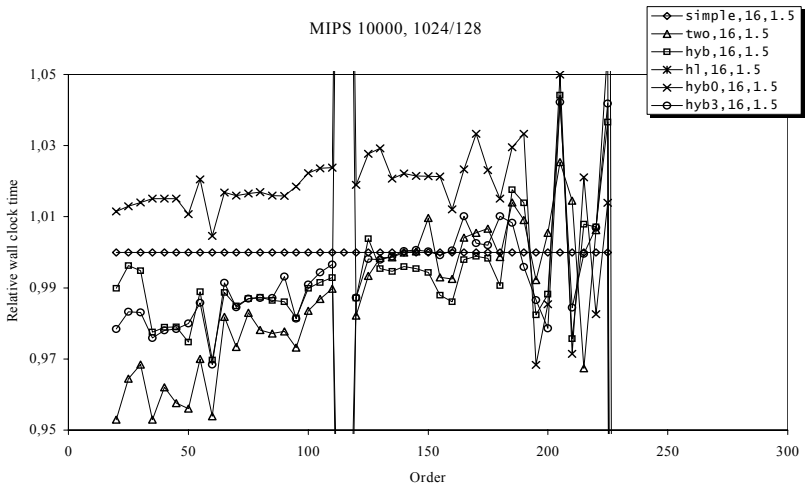


Chart C-39. Basic mergers, $(\alpha,d) = (16,1.5)$, MIPS 10000.

We conclude that the branch prediction unit of the Pentium 4 is very effective and that using any explicit intelligence to aid in avoiding slightly unpredictable branches will only hurt performance. The MIPS only has a six-stage pipeline, so any unpredictability in branches will not influence performance much. Still, it benefits from the tighter loop. Handling cases where the output buffer sets the limit on the number of elements merged in a special tight loop will improve average performance 3-5% percent on Pentium 3 and MIPS. Any more overhead and performance will get poorer.

Higher Order Mergers

Having established good ways to merge two streams, we are interested in extending the capability to merging of higher order and establish how that affects performance. [ACV+00] provides compelling evidence that merging with low orders can significantly increase performance; instead of using a traditional multiway mergesort, they restrict the sort to only use mergers of order no higher than some constant, instead of allowing the order to grow to M/B. This in turn will give them more passes, but the benefit of

using small mergers outweighs that cost. We are looking at the other end of the scale, comparing binary merge to higher order; never the less, we should see that performance increases, when increasing the order to a certain number.

There can be at least two reasons for any increase in performance. One is, as [ACV+00] argues, that merging e.g. four or six streams can be done with all stream pointers stored in registers. The same is the case with merging two streams but with more streams, the registers are better utilized. This will not be the case on the Pentium machines, where only eight general purpose registers are available, barely enough to hold the pointers involved in merging two streams, however spilling the pointers to fast L1 cache may not be a performance problem. The second reason that performance would benefit is that we skip potentially expensive tree navigation operations; using four-way basic mergers is like using two-way basic mergers, except the edges containing the smallest buffers have collapsed.

On the other hand, leaving two-way basic mergers also means leaving a compiletime knowledge of how many input streams a basic merger can have; using z-way basic mergers means we have to be able to handle merging of any number between two and z streams, since any of the z input streams may have become exhausted.

Let us examine the ways in which we can implement z-way basic mergers. Recall that a basic_merger implementation keeps a member variable active counting the number of non-empty input streams. A simple for-loop based extension of the simple_merger could then look like this:

```
template<class FwIt, class T, class Comp>
inline Token invoke(FwIt& b, FwIt e, Token outtoken, Comp& comp)
{
    struct ht { typename Stream::pointer h, t; } s[order];
    for( int i=0; i!=active; ++i )
        s[i].h = stream[i]->begin(), s[i].t = stream[i]->end();
    FwIt p = b;
    assert( active > 1 );
    while( p != e )
    {
        for( ht *m=s, *q=s+1; q!=s+active; ++q )
            if( comp(*q->h,*m->h) )
                m = q;
        *p = *m->h, ++(m->h), ++p;
        if( m->h == m->h ) // the input became empty
        {
            outtoken = token[m-s];
            break;
        }
    }
    for( int i=0; i!=active; ++i )
        stream[i]->begin() = s[i].h, stream[i]->end() = s[i].t;
    b = p;
    return outtoken;
}
```

Pairs of head and tail pointers are kept in an array on the stack. A for-loop finds the pair m with the head pointing the smallest element. The element is moved to the output and the head pointer incremented. This implementation is called simple_for_merger. An

implementation that, like the two_merger, uses a tight loop merging a minimum number of elements before recomputing the minimum is also implemented as the for_merger.

In the implementation above, each time we compare to find the smallest head, we do a double dereference. This can be alleviated by maintaining the value of the head along with the pair of pointers of that stream. However, this in turn means moving all elements to a temporary local variable, doubling the total number of elements moves. This could potentially be expensive when merging larger elements. for_val_merger and simple_for_val_merger has been implemented that are like for_merger and simple_for_merger, except they maintain a local copy of the head element.

A problem with all of these solutions is the overhead of the for-loop. While the stream with the smallest head can be isolated using conditional moves, neither the compiler nor the processor at runtime have any idea of how many streams we need to consider. Instead, we could do a switch on active out side the loop. In the switch, we now know what active is. The implementation simple_comp_merger uses this information as a template argument that then picks out the smallest head. The templates are illustrated here:

```
template<int active>
inline bool move_min(It *head, It *tail, Token *tokens, It out)
{
    if( *head[0] < *head[active-1] )
        return move_min<active-1>(head,tail,token);
    else
        return move_min<active-1>(head+1,tail+1,token+1);
}
template<>
inline bool move_min<2>(It *head, It *tail, Token *tokens, It out)
{
    if( *head[0] < *head[1] )
    {
        *out = *head[0], ++head[0];
        return head[0] == tail[0];
    }
    else
    {
        *out = *head[0], ++head[0];
        return head[0] == tail[0];
    }
}
```

and used like this:

```
switch( active )
{
    case 2:
        …
    case 4:
        for( p!=e; ++p )
            if( move_min<4>(head,tail,token,p) )
                break;
        break;
    case 5:
        …
}
```

In move_min<k> a comparison is made to see which of the first or the k-1st stream contains the larger element. That stream cannot contain the smallest element, so move_min<k> calls recursively on all but that particular stream. Provided the compiler inlines the entire recursion, this implementation will do exactly z comparisons per element merged in a z-way basic merger, and when they are done we know exactly where on the stack the pointer to the smallest element is. The problem is that the code is exponential in size and that none of the outcomes of the z comparisons are predictable nor can they be replaced by conditional moves. A version using minimum determination like the two_merger has also been implemented and is called comp_merger

Instead of using sequential comparisons, we can also use optimal data structures such as heaps. The looser_merger is based on a looser tree that only does $\log z$ comparisons and moves [Knu98]. It too has been implemented using templates; when the looser has been located, we switch on the number of its associated stream. In this switch, we call a function specialized for that particular stream which then updates the looser tree.

For the evaluation of the different implementations, we use them in a 120-funnel with $(\alpha,d) = (16.0,\ 2.0)$ to merge 1,728,000 elements. We do this eight times and measure the total time on a physical clock. For reference, we also include the binary basic mergers from the previous section. Here is the result:

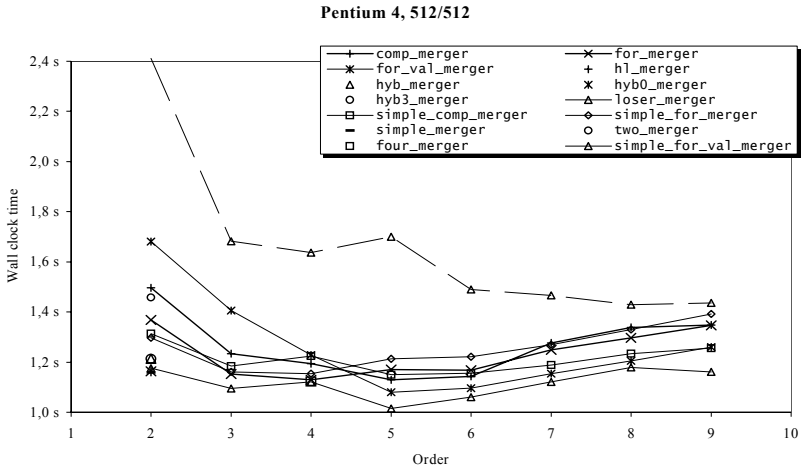


Chart C-40. Basic mergers, Pentium 4.

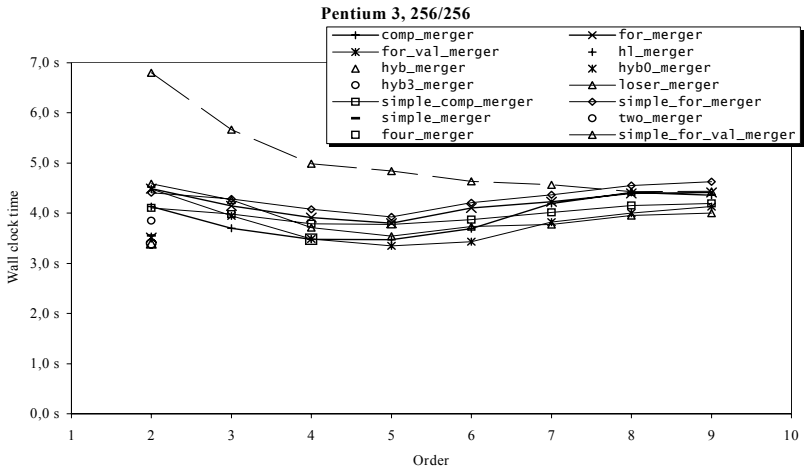


Chart C-41. Basic mergers, Pentium 3.

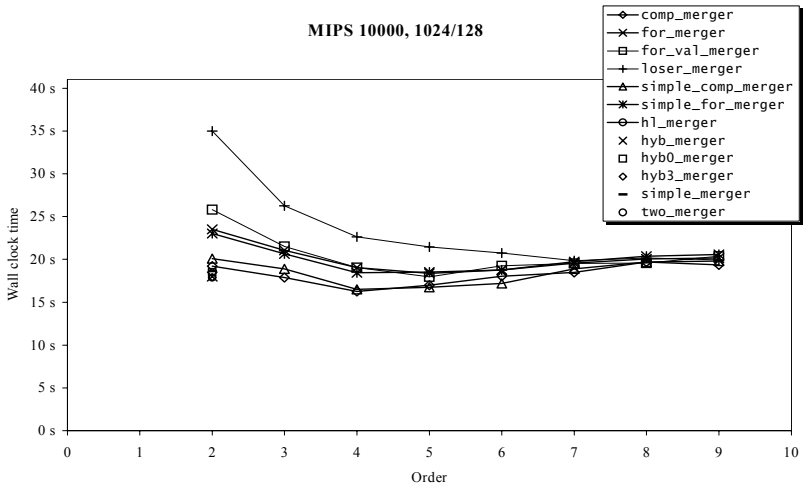


Chart C-42. Basic mergers, MIPS 10000.

Realizing compilers are not always eager to inline functions to the extend we need in the comp_merger and simple_comp_merger, we manually inlined a simple_comp_merger with $z = 4$. This is the four_merger. Since there is no discernible difference in performance between it and the simple_comp_merger, we conclude that the compiler does complete the inlining, at least for $z = 4$.

The charts clearly show there is performance to be gained from increasing z; however, at some point the performance begins to deteriorate. The optimum value seems to be either 4 or 5. The overhead of using the optimal loser_merger is too great to use on these orders. For sufficiently large z, determining the minimum number of elements merged and merging them in a tight loop is faster than the naïve approach. This could indicate that it is the small buffers in the tree that largely contributes to the overhead of this approach.

To some extent on MIPS but in particular on the Pentiums, it is hard to beat the handcrafted binary mergers. The reason for this is most likely the increased overhead of

making local copies of streams and iterating through them. Why the generalized simple_comp_merger takes such a performance hit when $z = 2$, compared to the simple_merger is not clear; the complier should inline the move_min<2> function and thus get a merge function identical to that of simple_comp_merger. As with the previous experiments, here too we must conclude that the simplest implementations are very good candidates to being the highest performer.

5.4 Funnelsort

Now that we have a high performing funnel in place, we will look into applying it in the algorithm for which it was designed. The algorithm as it is described in Algorithm 4-5, page 55, does not leave as many options open to the implementation. The analysis requires it to be recursive so we cannot experiment with the structure of the algorithm. However, there is a base case for which we need to decide how to sort and there is the matter of how the output of the merging should be handled. Finally, there is the matter of the values of α and d. We will first look at how to handle the output and memory management, introduce two final optimizations, then look at buffer sizes, and finally settle on the base sorting algorithm.

5.4.1 Workspace Recycling

In multiway mergesort (Algorithm 3-3, page 39), runs were merged using complete scans; the entire file of elements were read in and a file containing the merged runs was written to disk. The subproblems are solved in a level-wise order, allowing the reading and writing of all elements from and to disk at each level. The reason this is optimal is that the number of levels in the recursion exactly fits with what is possible with the block and memory sizes, namely $\mathcal{O}(\log_{M/B}(N/B))$.

The number of recursions in funnelsort will be higher ($\mathcal{O}(\log\log N)$) so we cannot merge by scanning all elements. We have to follow the recursion and store the output of one recursive call before we recurse to the bottom of the next problem and we cannot simply keep a file for each level in the recursion. One simple solution would be to, for each recursive call, allocate a buffer the size of the subproblems in that call, around $\alpha^{1/d}N^{d-1/d}$ elements. Each recursive sort would then put their output into that buffer and when the recursive sort was done, the elements of the buffer would be copied back into the original array. In this approach, providing the output space for the mergesort is left to the caller, making the interface look essentially like the std::copy STL function:

```
template<class Merger, class Splitter, class RanIt, class OutIt>
OutIt mergesort(RanIt begin, RanIt end, OutIt out);
```

The body would consist of allocating the temporary buffer and a number of recursive calls, each followed by a call to std::copy, to free the temporary buffer.

The problem with this approach is that all elements are merged to a buffer and copied back. That is one more move per element than need be made. We can do better than that, observing that when we have made the first recursive call, all the elements from that subproblem are now in the temporary buffer. That leaves a “hole” in the original array just big enough to hold the output of the next recursive call. When all the recursive calls have completed, the hole have moved to the end of the array. We then

do the only move of elements, namely from the buffer to the end of the array. The process is illustrated in Figure 5-8. Using this procedure saves us a considerable N-$\alpha^{1/d}N^{d-1/d}$ element moves.

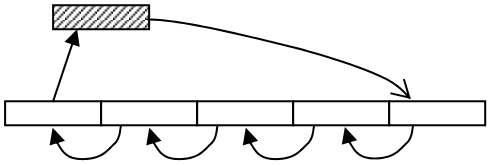


Figure 5-8. The merge procedure used in funnelsort. The thick arrows indicate sorting output while the thin arrow indicates a move.

In each recursive call, we need the temporary buffer and a k-funnel, but not both at the same time. Using the stack_allocator, described in Section 5.3.2, we can first compute which of the two takes up most space, construct a stack_allocator large enough to hold either of them, allocate the buffer, sort recursively, move the buffer elements back into the array, deallocate the buffer, and then layout the funnel using that allocator. This way, the funnel is laid out in exactly the memory locations the temporary buffer occupied. Recycling the workspace like this, will likely mean that the funnel is already in cache when it is needed.

5.4.2 Merger Caching

As with any function, at each recursive call a new set of local variables are allocated and constructed on the execution stack. This is normally not much of a performance issue, but if one of those variables is a funnel, having to allocate and construct it at each recursive call may soon become a performance issue.

In fact, constructing a new funnel at each recursive call is far from necessary. In all calls at the same level of recursion, we use a funnel of the same order, so instead of using a funnel local to the merge_sort function, we start out by simulating the recursive calls of the merge_sort function and at each level noting the order of the funnel needed. A funnel is then allocated for each level and they are in turn used in the recursive calls. For the simulation, we are only interested in the levels of the recursion and so could do with a single tail-recursive call, easily converted to a loop.

Using this scheme, we can only apply workspace recycling at the root of the recursion, but since that will dominate the rest of the recursion, both in workspace consumption and memory transfers, this will also be where we gain the most.

We have implemented the funnelsort algorithm both with and without merger caching to asses whether pre-computing the total space needed throughout the algorithm will be a considerable overhead, or constructing a new funnel in each call will hurt performance. The premise of using workspace recycling was that the funnel used the same stack based allocator as used to allocate the temporary buffer. However, since we deallocate the buffer just before we start allocating the funnel, using a heap allocator could achieve the same effect, if the allocator chooses to allocate from the newly freed area. At the same time, using a heap allocator may be slower than the stack_allocator, due to the complexity of managing a general heap, thus shifting the performance in favor of using merger caching.

We have tested the two versions of funnelsort with both a stack_allocator and a heap allocator. For this test we use $\alpha = 4$, $d = 2.5$, and the simple_merger basic merger ($z = 2$). We use the std::sort provided with STL to sort subarrays smaller than $\alpha z^d = 23$. We sort uniformly distributed pairs. The result is as follows:

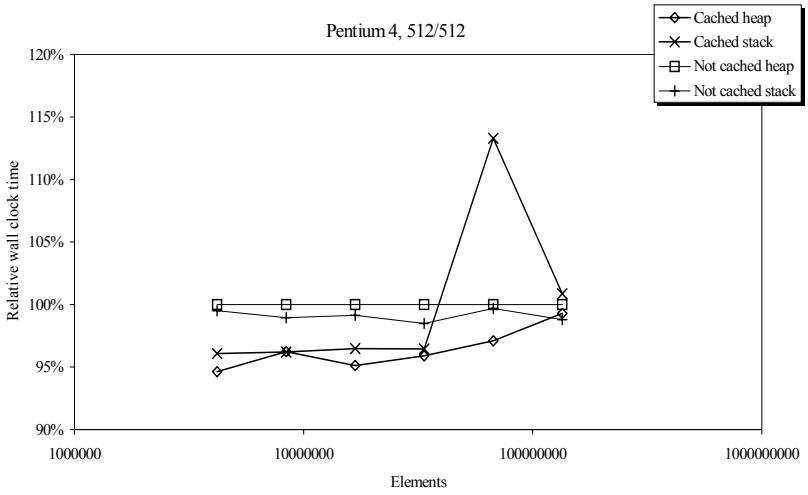


Chart C-43. Effects of merger caching, Pentium 4.

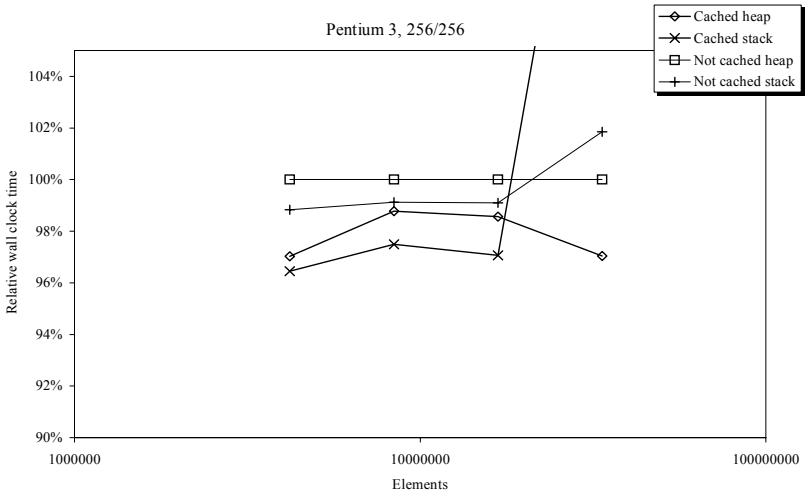


Chart C-44. Effects of merger caching, Pentium 3.

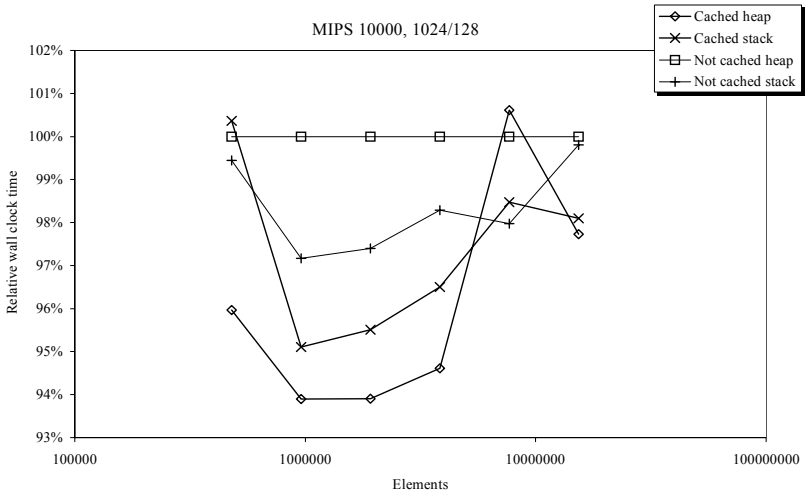


Chart C-45. Effects of merger caching, MIPS 10000.

We saw in Sections 5.3.2 and 0 that the different architectures preferred different allocators. We see the same picture here. We do however see a more consistent picture here; all architectures clearly prefer the mergers to be cached. We suspect that this is mostly due to avoiding the computational overhead of constructing mergers in each recursive call. There is only slight evidence of savings due to increased locality, by recycling workspace and using std::allocator, as can be seen here in the number of TLB misses:

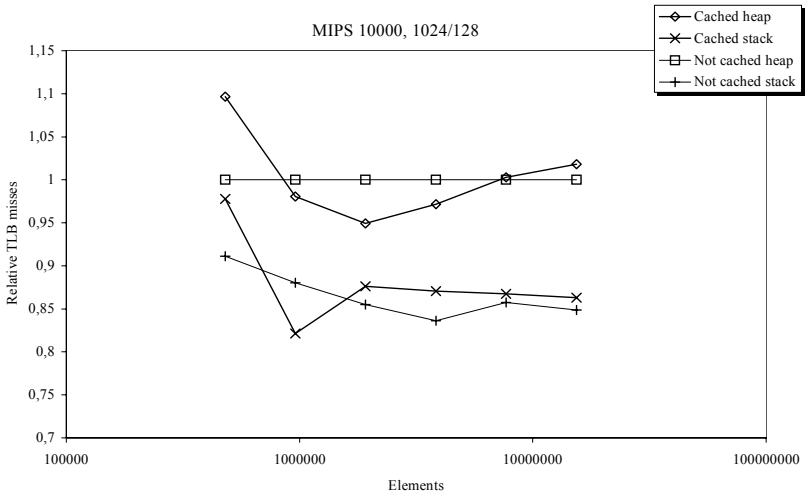


Chart C-47. Effects of merger caching, MIPS 10000, TLB misses.

The effect of reusing mergers is dwarfed by the effect of using stack_allocator. As discussed, using the stack_allocator allows us to reuse the temporary buffer for laying out the funnel. When we recycle the workspace like this, we effectively recycle virtual memory addresses, in turn keeping the translation look-aside buffer entries alive longer. This reduction in TLB misses does not affect the overall execution time significantly,

however. We should use both merger caching and controlled allocation to reduce the construction overhead and increase overall locality.

5.4.3 Base Sorting Algorithms

As an improvement to quicksort, Sedgewick introduced the idea of not completing the quicksort recursion, but stop before problem sizes got too small [Sed78]. This would leave the elements only partially sorted. To sort it fully, insertion sort was used in a final pass. What made it efficient was the special property of insertion sort, that if no element is more than c places from where is should be in the sorted sequence, insertion sort can sort all n element using no more than $\mathcal{O}(cn)$ moves and comparisons [Knu98]. Ladner and LaMarca have since proposed that the insertion sort should be done at the bottom of the recursion rather than as a final pass, since a final pass would incur N/B additional memory transfers [LL99]. As a side effect, the special property of insertion sort is no longer needed; any low instruction count sorting algorithm can be used.

With funnelsort, we are faced with a similar situation – below a certain problem size, we have to switch to a different sorting algorithm, simply because no funnel can merge such small streams. We choose to switch to another algorithm when problem sizes becomes smaller than αz^d, because that in turn will make funnelsort choose at least a $z+1$-funnel, that is a funnel of greater than one height, on all inputs sorted by funnelsort. This avoids the need to handle the special case, where the root of a funnel is also a leaf.

The choice of sorting algorithm for the base is not clear. Insertion sort as proposed by Sedgewick performs $\mathcal{O}(n^2)$ moves in the worst case; however, it performs much better when applied to data that is almost sorted. Indeed, it naturally detects completely sorted sequences with only $\mathcal{O}(n)$ comparisons and uses no moves at all. A very low-overhead alternative to insertion sort is selection sort [Knu98, Algorithm S]. A compelling feature of selection sort is that for each position in the sequence, the correct element is located and then moved there; it only moves an element once. However, it does $\mathcal{O}(n^2)$ comparisons even in the best case.

The limitation of insertion sort is that most elements are never moved more than one position. Shell sort attempts to remedy this by doing several passes of insertion sort, first only on elements far apart, then on elements closer and closer to each other [Knu98, Algorithm D]. It has a higher overhead but will asymptotically perform fewer operations per element. Considering that modern processors are super-scalar and capable of executing several instructions in parallel, it is only natural to investigate sorting algorithms that are not inherently sequential. One such algorithm is Batcher's merge sort [Knu98, Algorithm M]. Similar to Shell sort, Batcher's sort uses several passes, each sorting elements closer and closer together. The difference is that the sequence of comparisons in Batcher's sort is such that they can be executed in parallel. Modern processors may be able to detect and exploit this. The downside is that computing the sequence gives this algorithm a considerable overhead. Heapsort is a special kind of selection sort, where each element is selected in $\mathcal{O}(\log n)$ moves and comparisons, making it an asymptotically optimal sorting algorithm.

These algorithms were implemented and run on small arrays of uniformly distributed random pairs. We measure the wall clock time it takes to sort a total of 4,096 such pairs. For this test, we had the unique opportunity to run on an Intel Itanium 2-based computer. The Itanium class of processors uses so-called explicit parallelism. This means that when the compiler issues instructions, it will bundle them in

instructions capable of being executed in parallel. This is opposed to RISC and CISC architectures, where instructions are emitted by the compiler as sequential as they should be executed and the compiler is not concerned with what instructions can be executed in parallel. It will then attempt to extract any parallelism. Another side of the Itanium architecture is the heavy use of conditional execution; all instructions can be executed conditionally and on any of 128 predication bits. The results are as follows:

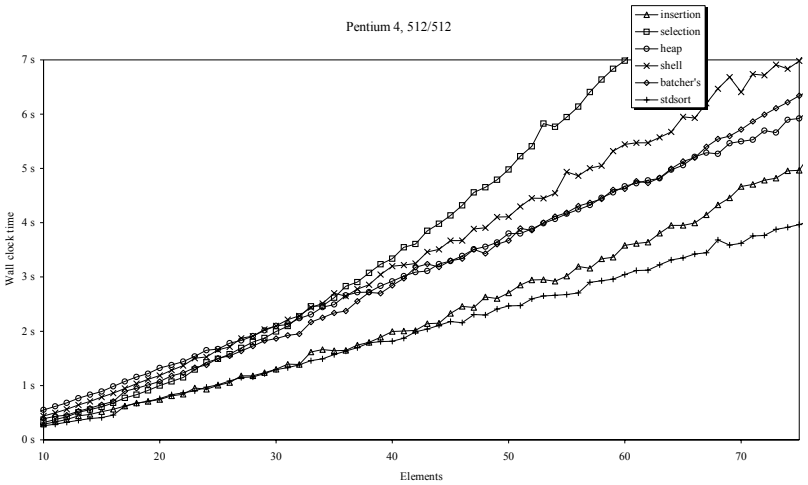


Chart C-48. Base sorting algorithms, Pentium 4.

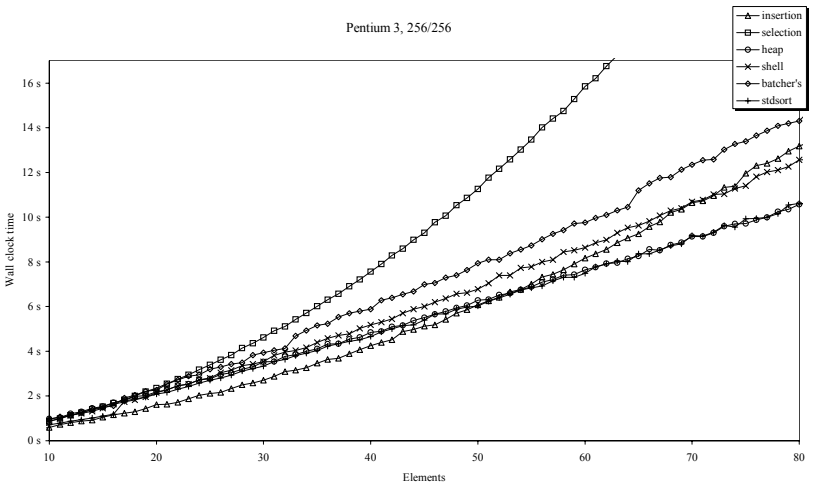


Chart C-49. Base sorting algorithms, Pentium 3.

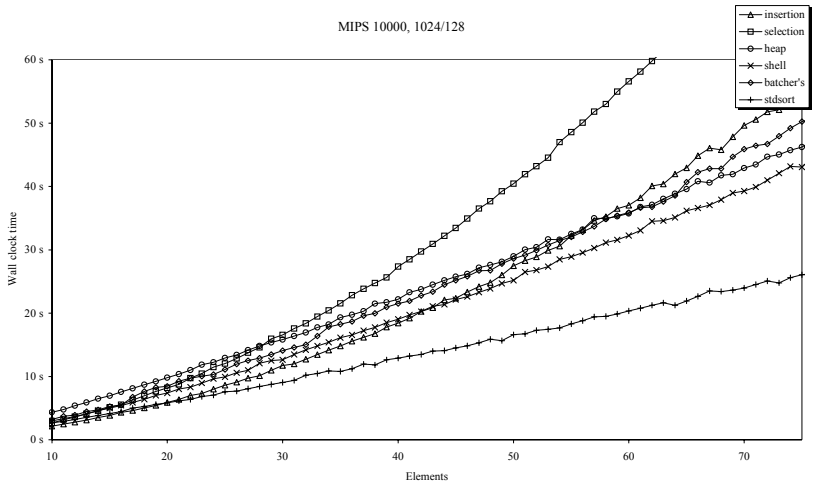


Chart C-50. Base sorting algorithms, MIPS 10000.

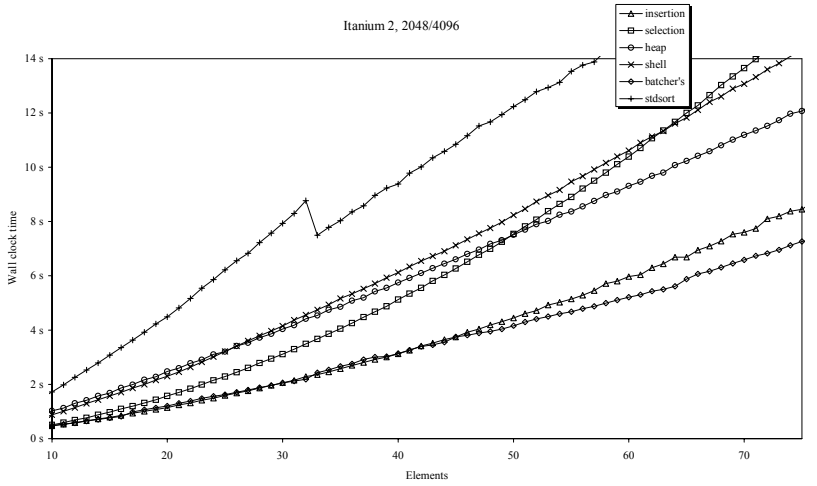


Chart C-51. Base sorting algorithms, Itanium 2.

For the test on the Itanium, we used the Intel C++ compiler version 7. This compiler comes with the Dinkumware implementation of the STL. This particular implementation features an std::sort function that like the SGI implementation is based on introsort. However, for the partitioning, a more robust function is used than in the SGI implementation. This function does a so-called Dutch flag partition, collecting elements that are equal to the partition element between the two partitions. Furthermore, it uses a sophisticated rotate function in the implementation of insertion sort used in the bottom of introsort. In all, while it makes the implementation faster on certain inputs, it clearly makes it slower on the sets we tested. The switch to insertion sort is std::sort can clearly be seen. In the SGI implementation (perhaps most clear on the Pentium 3 results) the switch happens at 16 elements, while in Dinkumware, it happens at 32 elements. Sedgewick originally suggested a switch around 9 or 10 elements; however, we see here that insertion sort remains competitive at least up in the 20's, even 40's on the Pentium 4, at least when sorting uniformly distributed pairs.

Selection sort is apparently too hampered by its best-case $\mathcal{O}(n^2)$ comparison count to be competitive. Comparing selection sort to insertion sort, we can see that insertion sort is indeed significantly faster than its $\mathcal{O}(n^2)$ worst-case time. Eventually, however, insertion sort will loose to all but selection sort. The optimal heapsort is quite competitive on all architectures and most problem sizes, while some architectures prefer Shell sort more than others.

Most interesting is perhaps Batcher's sort. On Pentium 3 and MIPS, its performance is in the mid-range, for the most part performing worse than Shell sort does. However, on Pentium 4, it performs better than Shell sort performs and is even able to keep up with heapsort. On the Itanium, however, it outperforms all other algorithms, being almost twice as fast as heapsort. This indicates that as processor performance get more and more dependant on instruction level parallelism, more and more performance can be gained when using sorting algorithm that allow for such parallelism.

As suspected, at least on the more traditional architectures, no algorithm can beat the hybrid and highly optimized approach of introsort.

5.4.4 Buffer Sizes

Finally, the implementation details of the complete funnelsort are in place. Without further ado, here are the results of sorting using funnelsort with different values of α and d:

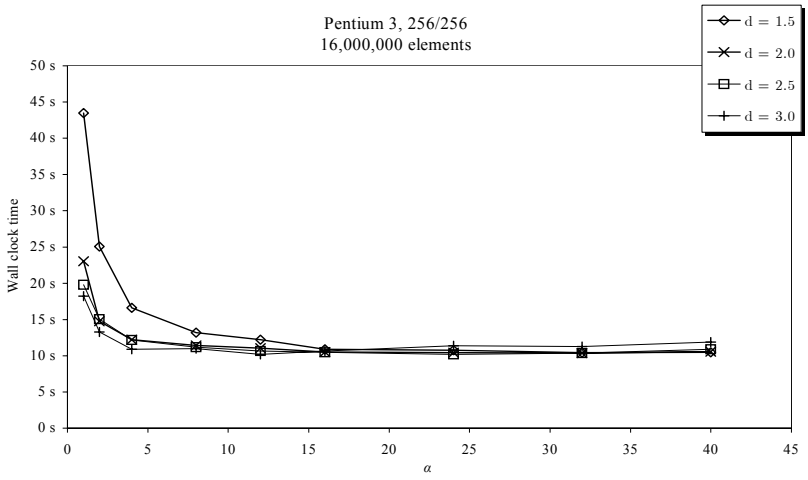


Chart C-55. Buffer parameters, sorting 16,000,000 elements on Pentium 3.

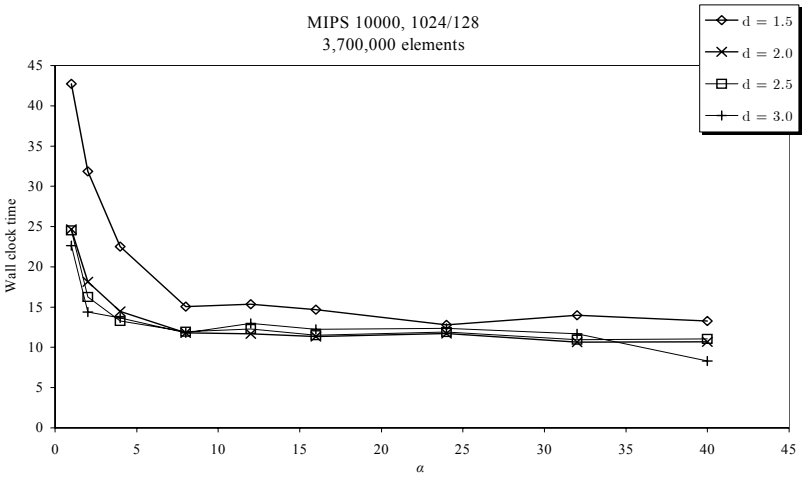


Chart C-58. Buffer parameters, sorting 3,700,000 elements on MIPS.

The test was conducted for three different array sizes on each machine, all of which fit in main memory. As suspected, when decreasing the values of α and d, fewer elements are merged per call to fill, and the overhead of navigating the tree and managing buffers become significant. With $\alpha > 4$ and $d \geq 2$, we can see that this overhead is virtually gone. Maximal performance is reached around $\alpha = 16$ and $d = 2.5$.

Choosing α and d is not as simple as the above two charts imply, however. The choice of values influences both the order of the funnel used and the space needed to hold it. To expose these effects, one of the array sizes were chosen close to what can fit in RAM. The results are as follows:

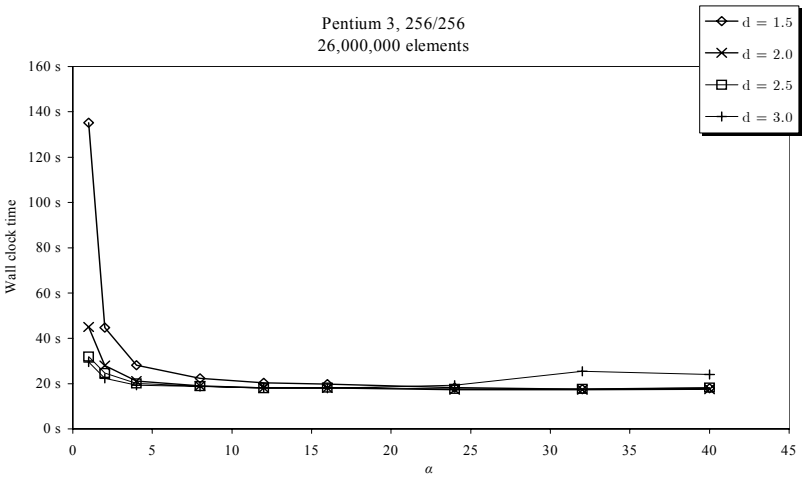


Chart C-57. Buffer parameters, sorting 26,000,000 elements on Pentium 3.

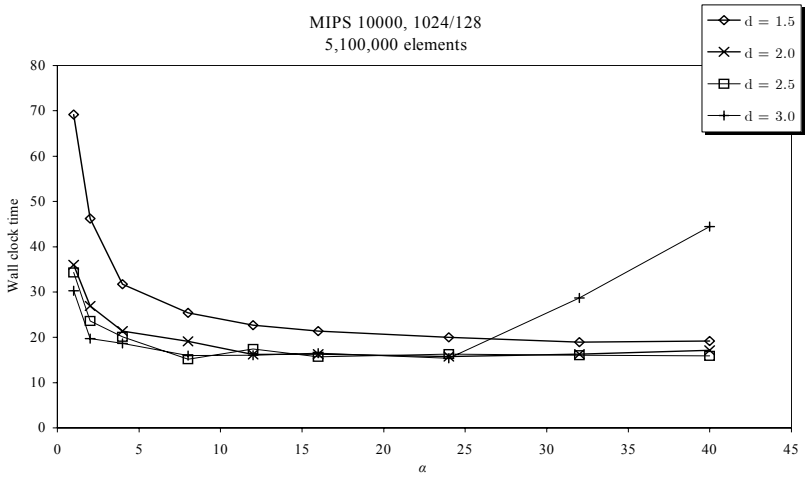


Chart C-59. Buffer parameters, sorting 5,100,000 elements on MIPS.

Two effects are dominating. One, if we choose d small, we will need a very high order funnel, and since $d > 1$, the total space consumed by its buffers are super linear. The total space needed for the algorithm then becomes too much to fit in memory. On the other hand, when the values of α and d are increased, the funnel it self will require more space and even though a lower order funnel is used, the size of the funnel is again too much for it to fit in RAM.

For any choice of values of α and d, the algorithm will require space for the funnel and for some array size this particular choice will make the total space requirements of the algorithm too high for it to fit in cache. The point is that we should avoid extreme values of α and d, since it will cause extreme space requirements of the funnel; it may be tempting to choose high values of α and d to minimize the overhead; however, doing so may cause the algorithm to incur memory transfers on smaller arrays than had we chosen more sensible α and d.

5.5 LOWSCOSA

The primary components of the LOWSCOSA are partitioning and merging with funnels. With a high performance funnel, already in place this leaves partitioning, which we will look at in this section. At the end of the section, we will briefly discuss what performance to expect from the LOWSOSA.

5.5.1 Partitioning

Partitioning elements of an array consists of two phases: median finding and partitioning. The partitioning phase uses the median as a pivot element and during a single scan moves elements that are larger than the pivot to one side and elements that are smaller to the other side. The exact median can also be found in linear time [BFP+73].

For algorithms like quicksort, we are not required to partition into two equally large partitions. For those algorithms, we thus do not have to use the exact median as the pivot; we can make due with an approximate median. Such a median can be computed

as the exact median of a small sample of elements instead of all elements. Popular sample sizes for quicksort are three and nine [Knu98], even $\sqrt{n}$ [MR01]. The effects of this is that virtually no time is spend finding the median and thus only the partitioning contributes to the linear term in the complexity. A downside is that we risk making uneven partitions where one part not much smaller than the original array. This can mean that the time spent partitioning is largely wasted. In quicksort, however, even with a sample size of one, that is we use a predetermined element as the approximate median, on uniformly distributed elements the expected running time is only a small constant larger than what could be achieved if we new the exact median in advance [Knu98].

In the interest of performance, we would like to use an approximate median for the LOWSCOSA also. The consequences of the resulting uneven partitions are however not as trivial as in the case of quicksort. If we partition such that there are more small elements than large elements, the output of the merger cannot fit in the space originally occupied by the large elements (see Figure 4-3, page 58). This is a design problem in the algorithm that needs to either be solved or avoided. This means that we cannot hope to generate more sorted elements than there are elements in the smaller of the two partitions at each iteration of the LOWSCOSA. Furthermore, if we go ahead, sort the large number of small elements for the input to the funnel, and only output a small number, we have wasted a considerable time sorting them.

5.5.2 Strategy for Handling Uneven Partitions

Before we look at how to handle the case of an uneven partition, we make the following observation. It is possible to combine the partition phase with the sorting of the subarrays to be merged in the current call; during the partitioning, when we have moved a sufficiently number of small elements to the end of the array, we put the partition on hold and sort them. This way, when these subarrays are small enough to fit in cache, we can complete the partition phase *and* the following sorting phase incurring only N/B memory transfers instead of up to $3N/(2B)$. We consider this an important optimization in the interest of increasing locality and cache usage.

Repartition

The simplest strategy is perhaps to perform the partition using the approximate median. When that is done, if more than half the elements we partitioned as smaller than the median, we simply pick a new median and partition again. An improvement is to only partition the small elements. This will take less time and more likely generate a partitioning with fewer small elements than large.

However, this approach makes it infeasible to sort streams while partitioning, because we risk having to repartition and thus make the sorting a wasted effort. In addition, we would expect every other partition to generate more small elements than large, so repeating the partitioning every time that happens will generate a considerable overhead.

Abort on Empty Refiller

Instead of repartitioning, thus avoiding the problem of outputting too many elements, we can continue with the uneven partition and handle the problem explicitly. The problem can be solved by giving the refiller a way to abort the merging. It would then

do so just before it begins reading into the part containing small elements. The situation is depicted in Figure 5-9. This will leave a "hole" in the input between the output and the refiller. The hole is patched with the elements contained in the funnel and included in the recursive call.

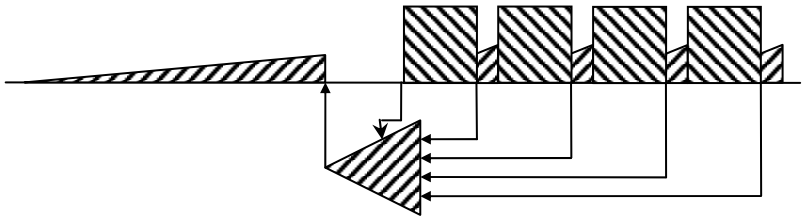


Figure 5-9. The refiller has no more large elements.

This scheme avoids a second partitioning and allows us to sort the input streams during the partitioning; however, it is flawed in the case of extremely few large elements. In these cases, the buffers in the funnel will not be filled and not a single element output. The refiller reads in all large elements before a single element is output from the funnel. In these extreme cases, we would have to fall back on the repartitioning scheme or employ some other special-case handling scheme.

Abort on Full Output

To remedy the fault, we may continue the merging until we have filled the left side of the array with small elements. To avoid the refiller starting to read in parts of the sorted streams, potentially duplicating elements, we need to detach it from the funnel. We have to keep the input streams attached so the funnel keeps reading the small elements. When the output has filled the left side, some elements are both in the input streams (the space they occupied was not refilled) and in either the funnel or in the output. In essence, the hole from the previous scheme is now scattered in all the input streams. Like before, we fill these holes with the elements remaining in the funnel.

Merge Big Elements

Perhaps the most elegant approach is to make input streams of the elements in then smallest of the partitions, not necessarily of the small elements. If the smallest of the partitions contain large elements, we use a funnel that outputs large elements first and writes them from the end of the array to the beginning. The process is illustrated in Figure 5-10. Note that the output and input of the funnel is now written and read in reverse direction.

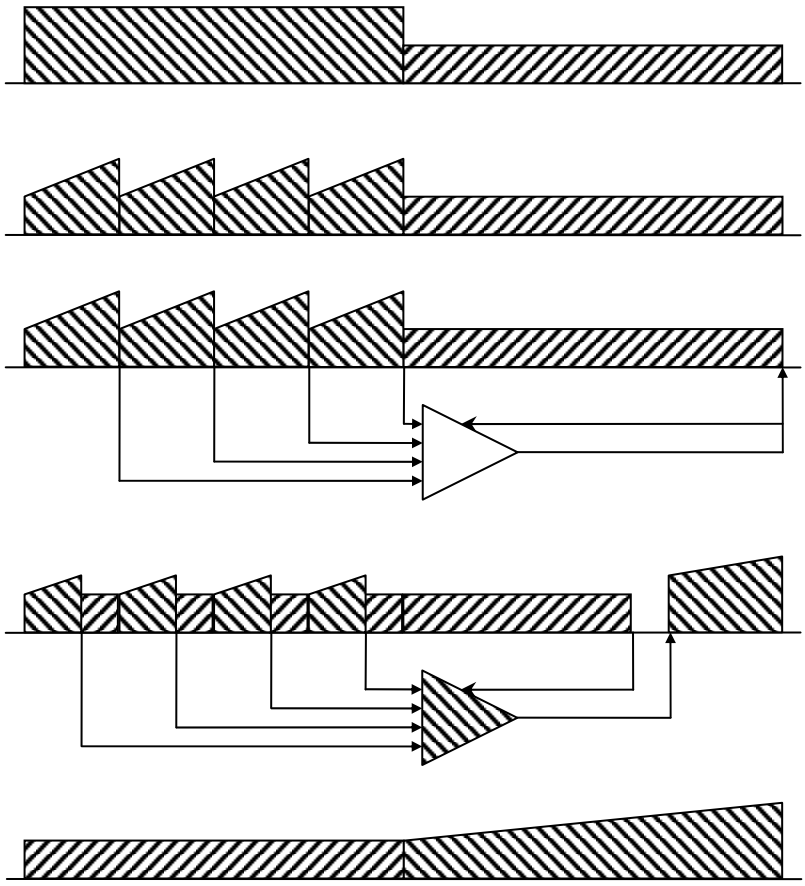


Figure 5-10. The process of multiway merging when there are more small elements than large element after the partition.

In our implementation, we have chosen to detach the refiller when it hits the small elements and to sort the streams while partitioning. This means that if we do an uneven partition, we may have sorted streams containing many more elements than the number of elements sorted by the end of an iteration. To reduce the risk of that happening, we have increased the sample size to 31 elements. We sort this sample and use the 17^{th} smallest as the partition. Using a larger element than the 15^{th} smallest reduces the risk of partitioning more small elements than large elements.

5.5.3 Performance Expectation.

When looking at the virtual memory level of the memory hierarchy, we have argued that for all sensible input sizes neither multiway mergesort nor funnelsort will do more than $4N/B$ memory transfers. The LOWSCOSA will unfortunately do more than that.

Under the assumption that an entire input stream of the funnel can fit in memory, the partitioning and sorting of input streams can cause up to $2N/B$. The funnel will read in and write out half the elements for a total of N/B memory transfers. However, by now only half the elements are sorted, assuming partition into equally many large and small elements. The algorithm will continue on arrays of geometrically decreasing sizes, effectively doubling the the number of memory transfers. The total number of memory transfers will be largely dominated by those incurred at the first iterations, so when N is

significantly larger than M, the fact that the last $\log M$ iterations can be performed without incurring memory transfers has little influence.

Thus, the total number of memory transfers incurred under these assumptions could be as high as $6N/B$. With an input occupying 2GB and half a gigabyte of RAM, the first two iterations incur the full number of memory transfers, while after the partitioning phase of the third iteration the rest of the algorithm activity is within RAM. This gives a total of $(3+(3/2)+2/4)N/B = 5N/B$ memory transfers, only slightly less than quicksort on the same input size and under more realistic assumptions (see Section 3.2.4).

In addition to added memory transfers, the LOWSCOSA also has an increased instruction count; each time an element is read into the funnel, another element is moved in its place. This will, in turn, almost double the total number of element moves performed. The number of comparisons is also increased, since before an element is in its right place, it has participated in a partitioning and a merge, as well as the recursive sorts. Furthermore, the LOWSCOSA requires $\mathcal{O}(\log n)$ funnels to be constructed as opposed to funnelsort requiring $\mathcal{O}(\log\log n)$ funnels.

In all, the expectation of the performance of the LOWSCOSA does not look good. It is however optimal in the cache-oblivious model and the fact that it requires low order working space is for us reason enough to investigate its performance.

Chapter 6

Experimental Results

We will now apply our sorting algorithms in an experimental study comparing it to other sorting algorithms. The goal is to establish whether cache-oblivious sorting algorithms can compete with simpler sorting algorithms designed to be efficient in the RAM model and with algorithms tuned to be efficient in the use of caches.

We start with an overview of how the study is conducted and then present the results.

6.1 Methodology

6.1.1 The Sorting Problem

When comparing different algorithms that solve the same problem, we need to take care that they are treated equally. To do this we will clearly define the problem we wish to solve, and do it in a way, which does not favor any algorithm. The problem we will look at in the next to sections is as follows.

We are given the name of a file stored on a local disk on a native file system. The file contains a number of contiguous elements. No part of the file can be in memory beforehand. The problem is solved, when a there exists a file in the file system with the same elements but in non-decreasing order stored contiguously. We do not require the file to be physically stored on disk, nor do we require the elements to be in the original file.

The reason we use the file system and state explicitly that no part of the file may be in memory when the algorithm is started is that different algorithms access elements in different order. Mergesort typically access elements from the first to the last, while partition based sorting algorithms access elements from the both ends of the file. If we were to generate problem instances by writing elements to disk, in a streaming fashion, the last M elements of the file would likely be cached in memory. This means that partition-based algorithms that access the last elements early will have an advantage, because these elements can be accessed without causing memory transfers. In general, we do not believe such an advantage to be present in real life problems.

To ensure that no elements are in memory, when the sorting begins, we run a small program, named fillmem, after the input file has been generated. fillmem allocates an array exactly the size of main memory. It then writes values in all entries of the array, forcing the operating system to allocate pages for the array and evicting pages already in memory. Typically an operating system prefer to evict pages occupied by the file

system cache before evicting pages used by user processes [Tan01], so we expect this process to make sure no part of the input is in memory.

In the Datamation Benchmark, the input is also in a file on disk [DB03]. However, they require the output to be stored in a different file. If we were to require this from all algorithms, we would have to add an artificial copy phase to all algorithms that sort in-place, thus giving the merge-based sorting algorithms an artificial advantage. We expect most applications of sorting algorithms to want only the sorted elements and not to care how or where they are stored.

6.1.2 Competitors

There are several different classes of competitors to choose from, when comparing sorting algorithms. We wish to compare with algorithms known to be efficient due to low instruction count and with algorithms that are efficient due to efficient use of the memory hierarchy.

Cache Optimized Algorithms

LaMarca and Ladner implement sorting algorithms optimized for L1 or L2 cache [LL99]. Improving on their effort, Wickremesinghe *et al.* implements sorting algorithms, called R-merge and R-distribution, which utilizes registers and lower level caches better [ACV+00]. Kubricht *et al.* implements variants of the algorithms of LaMarca and Ladner that also takes the translation look-aside buffer and low associativity into account [XZK00].

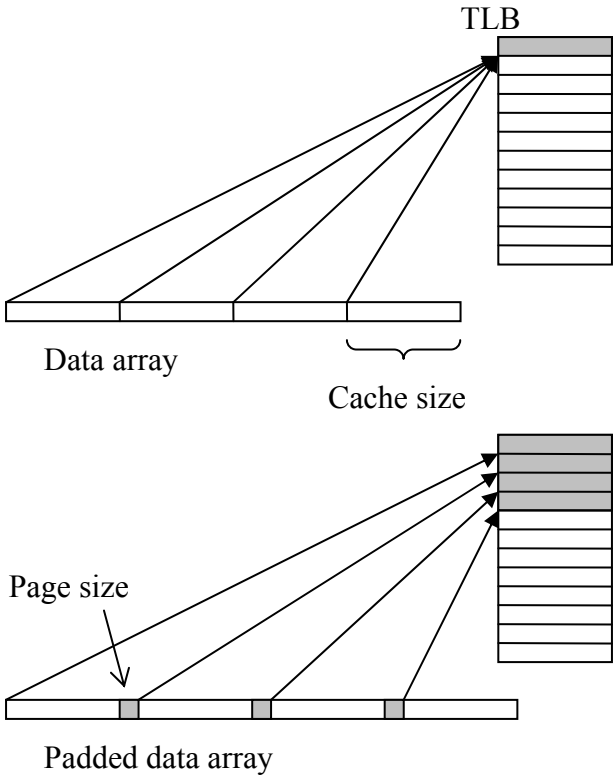


Figure 6-1. Multiway merging with and without padded inputs.

We would have liked to compare our algorithm to those of [ACV+00] and [XZK00]. However, the source code for R-merge and R-distribution is not publicly available and we are still waiting for a reply to the request to obtain it. The source code used in [XZK00] is publicly available. However, we did encounter problems using it.

The algorithms are based on LaMarca and Ladner's multi-mergesort and tiled mergesort. Multi-mergesort is essentially the multiway mergesort. It forms runs the size of the cache, sorts them while they are in cache, and merges all runs in a single pass using a heap. Tiled mergesort also use a run formation phase; however, it merges the runs formed using binary merging over several passes. The observation made in [XZK00] is that elements that are roughly one cache size apart are often mapped to the same TLB entry, since the TLB cover the same range of virtual address as the L2 cache (see Figure 6-1 above). This is in turn likely to cause a conflict miss on every element merged. For example, the Pentium 3 has 64 entries in its TLB, each covering 4KB for a total of 256KB, which is exactly the size of the L2 cache. The solution consists of introducing holes in the array containing the elements to be merged. These holes pad the runs formed during the run formation phase of the algorithms, so runs are separated by one page. If elements are read from each run at the same rate, this will insure that no conflict misses will occur.

In the implementation of padded tiled mergesort, the runs formed in each pass remain padded and the algorithm stops with the holes are still in the array. This means that this implementation does not solve the problem stated above. We do not consider an algorithm not generating an output of contiguous non-decreasing elements a valid sorting algorithm and thus exclude it from our benchmarks. The padded multi mergesort merges in a single pass and thus does not need to maintain the holes. However, we cannot confirm that it generates correctly sorted output and we believe the implantation to be buggy. The non-padded variants of tiled mergesort (`msort-c`) and multi mergesort (`msort-m`) seem to function fine, albeit only on input sizes of a power-of-two.

For our tests, we have changed their implementation slightly. We have changed the type of elements sorted from `long long` to a template parameter. Furthermore, the tiled mergesort did an explicit copy of the elements back to the original array, in case the array was not the output of the final merge pass. Since we do not require the elements to be in their original array, we have removed this final copy pass.

The algorithms are cache-aware and thus needs to know the size of the cache. For the Pentium builds, we specify a cache size of 256KB and for the MIPS build, we specify a cache size of 1024KB.

External Memory Sorting Algorithms

For algorithms designed for external memory, there is the LEDA-SM [CM99] and TPIE [TPI02].

The LEDA-SM is build on top of the commercially available LEDA library. We were able to get a copy of the LEDA library and the source code for the LEDA-SM is freely available. However, our copy of the LEDA library is designed for the GCC version 3 and LEDA-SM is written in a pre-standard dialect of C++ that GCC version 3 does not understand. We succeeded in translating most of the LEDA-SM into standard C++, only to realize that the namespaces used in LEDA made linking impossible. We

then got an older version of GCC and a matching version of the LEDA library, but when LEDA-SM still would not compile, we had to abandon the effort.

TPIE is also written in a pre-standard dialect of C++; however, unlike LEDA-SM, our old GCC compiler was able to build the TPIE library. TPIE includes a sorting algorithm, called ami_sort, that sorts a given input stream into a given output stream. These streams have to be managed by TPIE and cannot consist of files on disk. However, TPIE allows the streams to use explicitly named files, so we have created a program that generates TPIE streams as files on disk. These files can then be used in an actual sorting program as streams for the input of the sorting algorithm.

We only had the old version of GCC for Linux, so we will only be benchmarking AMI_sort on the Pentium computers. AMI_sort is also cache-aware and needs to know the amount of available RAM. The manual suggests specifying slightly less than the amount of physical memory, so we specified 192MB.

Sorting Algorithms for the RAM Model

The main competitor among classical sorting algorithms is the std::sort, available from the Standard Template Library accompanying any C++ compiler. The GCC compiler comes with the SGI implementation of the STL. This implementation uses the introsort by Musser for the std::sort function [Mus97]. Introsort is based on quicksort, but unlike earlier variants like [Sed78], introsort achieves a worst-case complexity of $\mathcal{O}(N\log N)$ and does so without sacrificing performance. This is achieved by detecting if the recursion becomes too deep, and if so switch to heapsort. Furthermore, the implementation uses insertion sort to handle problems of size less than 16, as suggested in [Sed78], however it is done at the bottom of the recursion, as suggested in [LL99], to preserve locality. For partitioning, it uses the fastest possible approach, not separating elements that are equal to the pivot element. We expect this implementation to very efficient and fully optimized.

Cache-oblivious Sorting Algorithms

For funnelsort, we use a variant of the funnel that is laid out according to the mixed van Emde Boas layout and uses the manually inlined four_merger as basic merger ($z = 4$). For computing buffer sizes, we use $\alpha = 16$ and $d = 2$. According to the study conducted in the previous chapter, these choices constitute a high performing funnel on all platforms. The LOWSCOSA uses the same funnel.

We have implemented a special output stream that uses the write system call (WriteFile in Windows) to generate the output directly on disk. It is implemented as a class containing a buffer with a capacity of 4096 elements. When the elements are written to this stream by funnelsort, the stream puts them in the buffer. When the stream iterator is incremented beyond the capacity of the buffer, it is written to disk and the iterator set to the beginning of the buffer. Using this stream, we avoid having to hold the entire output array in the address space, thus allowing to potentially sort up to 2GB of data using funnelsort. The variant using this stream is called ffunnelsort, whereas the one writing to an array in memory is called funnelsort.

6.1.3 Benchmark Procedure

Except the ami_sort, none of the algorithms is designed to work with files on disk. Indeed, the algorithms of [XZK00] expect the elements to be in an array in memory. To

solve the problem as stated, we use the memory mapping functionality of the operating systems. Memory mapping works like ordinary paged memory except a specified file is used as backing store, not the swap file.

A program, sortgen, that generates inputs of a given type of elements, a given distribution, and a given size has been build. In addition, for each of the sorting algorithms, a separate executable has been build that takes the name of the input file as an argument and the name of an optional output file. All but the TPIE executable then maps these files into memory using memory mapping.

The procedure for benchmarking is then as follows. For each algorithm we the use sortgen (or the program using TPIE streams in case of ami_sort) to generate an input file. We then use fillmem to force the operating system to flush its file cache and run the executable containing the sorting algorithm.

A list of all algorithms used in the benchmarks can be found in Table 6-1.

Algorithm	Source	File access	Cache-aware	I/O optimal
ami_sort	[TPI02]	read/write	Yes	RAM-Disk
msort-c	[XZK00]	Memory map	Yes	No
msort-m	[XZK00]	Memory map	Yes	L2 cache-RAM
std::sort	SGI/GCC	Memory map	No	No
funnelsort	This thesis	Memory map	No	Yes
ffunnelsort	This thesis	Memory map/write	No	Yes
lowscosa	This thesis	Memory map	No	Yes

Table 6-1. Sorting algorithms used in comparative benchmarking.

6.2 Straight Sorting

For the first collection of benchmarks, we focus on uniformly distributed data and study the performance on different data types. We will use the three data types described in Section 5.1.3. A sample of 1,024 key values can be seen in Figure 6-2.

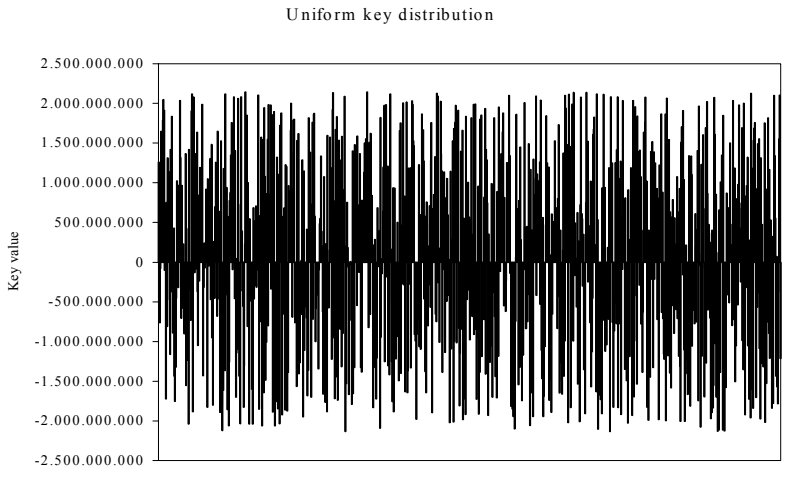


Figure 6-2. Uniform key distribution.

6.2.1 Key/Pointer pairs

The results of measuring wall clock time when sorting pairs are as follows:

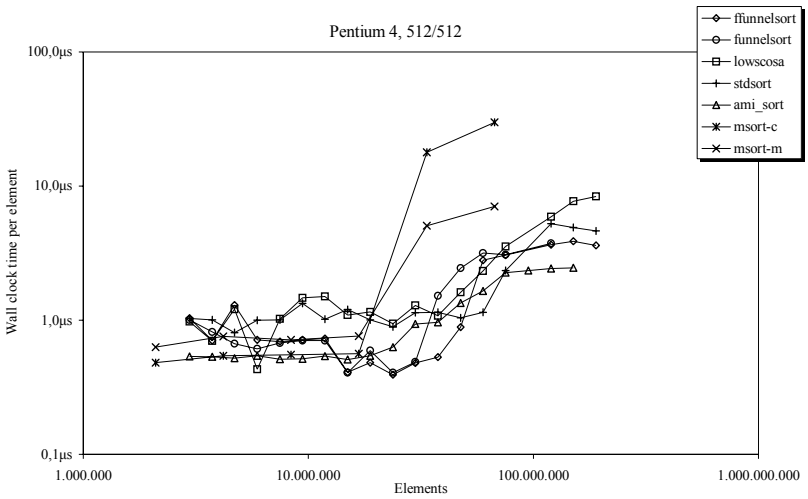


Chart C-61. Wall clock time sorting uniformly distributed pairs on Pentium 4.

The internal memory sorting algorithm used in TPIE is the fastest of the algorithms when the datasets fit in RAM. As a close second comes the cache tuned tiled mergesort. Relative to the tiled mergesort, the multi mergesort performs slightly worse. The reason for this is likely the overhead of managing the heap. While the cache-oblivious sorting algorithms cannot keep up with the memory tuned variants within RAM, do they outperform std::sort and perform on par with multi mergesort. As expected, the LOWSCOSA performs rather poorly.

The picture changes when the dataset takes up half the memory. This is when the merge-based sorts begin to cause page faults, because their output cannot also fit in RAM. The tiled mergesort suddenly performs a factor 30 slower, due to the many passes it makes over the data. We see that both funnelsort and TPIE begins to take longer time, however the LOWSCOSA and ffunnelsort does not loose momentum until the input cannot fit in RAM. Writing the output directly to disk instead of storing it, really helps in this region. The LRU replacement strategy of the operating system does not know that the input is more important to keep in memory than the output, so it will start evicting pages from the input to keep pages from the output in memory. When writing the output directly to disk, the output takes up virtually no space so the input need not be paged out.

When the input does not fit in memory, TPIE again has the superior sorting algorithm. This is indeed what it was designed for. It is interesting to see that it is so much faster than funnelsort, even though funnelsort incurs an optimal number of page faults. One explanation for this could be that TPIE uses double-buffering and overlaps the sorting of one part of the data set with the reading or writing of another, thus essentially sorting for free. Another explanation could be that is reads in many more blocks at a time. During the merge phase, usually no more than 8 or 16 streams are merged. Instead of reading in one block from each stream, utilizing only $16B$ of the

memory, a cache aware sorting algorithm could read in a much larger part of the stream, up to $M/16$ elements at a time. Both funnelsort and ffunnelsort outperform std::sort when the input cannot fit in RAM. This must be attributed to the optimal number of page faults incurred by the funnelsorts. Even though std::sort is, in some sense, close to being optimal, it is clear that it is not. The LOWSCOSA, unlike the funnelsorts and TPIE, does not seem to reach a plateau. This is because it is it keeps incurring a significant number of page faults due to it only sorting half the dataset per pass.

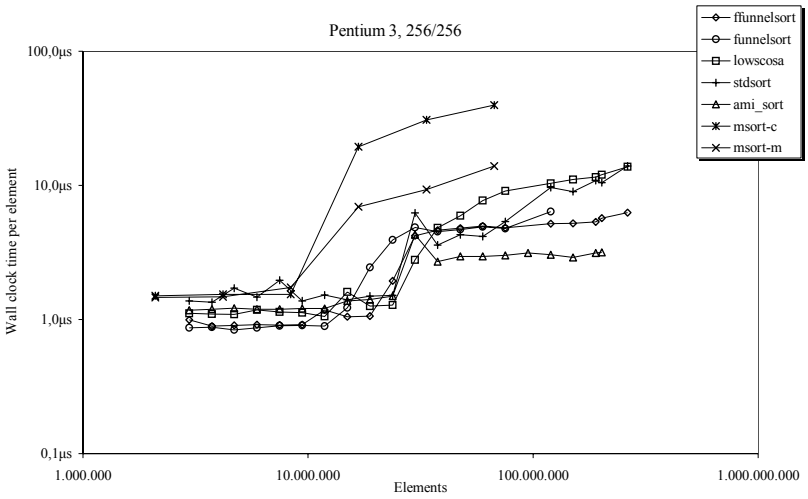


Chart C-62. Wall clock time sorting uniformly distributed pairs on Pentium 3.

On the Pentium 3, things are turned around a bit. Here, the funnelsorts are the fastest sorting algorithms when dataset fits in RAM. They outperform both the cache tuned algorithms and std::sort. Even the LOWSCOSA can compete with TPIE. Beyond RAM, we again see TPIE as the fastest sorter. As the dataset becomes much larger than RAM, we can see that funnelsort holds its performance level, while std::sort becomes slower and slower per element sorted. We can also see that the running time per element of the LOWSCOSA almost becomes constant, and that it will eventually outperform std::sort.

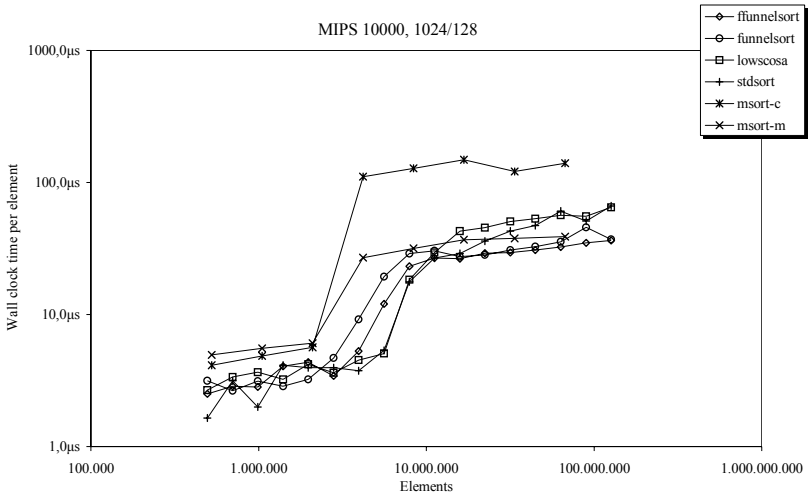


Chart C-63. Wall clock time sorting uniformly distributed pairs on MIPS 10000.

On the MIPS, the picture is not that clear, when looking at the time for sorting datasets that fit in RAM. However, we can see that the cache-tuned algorithms perform rather poorly. This is likely to be because of the many TLB misses they incur. The MIPS uses software TLB miss handlers, so the cost of a TLB miss is greater here than on the Pentiums. We see the same trend of the performance of std::sort, ffunnelsort and the LOWSCOSA not degrading until the input cannot fit in RAM. Then, we see funnelsort as the fastest sorting algorithm and the performance of std::sort continuing to degrade. As on the Pentium 3, the LOWSCOSA settles in with a somewhat higher running time than the funnelsorts but is eventually faster than std::sort.

Let us see, if we can locate the cause of these performance characteristics in the number of page faults incurred by the algorithms.

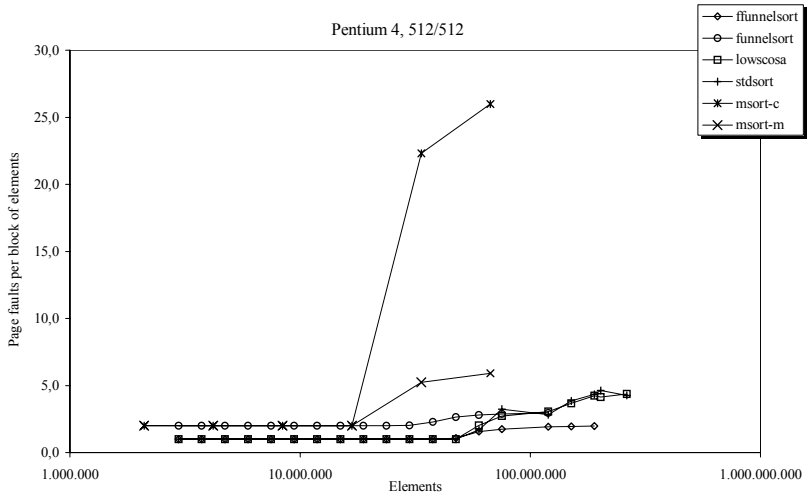


Chart C-64. Page faults sorting uniformly distributed pairs on Pentium 4.

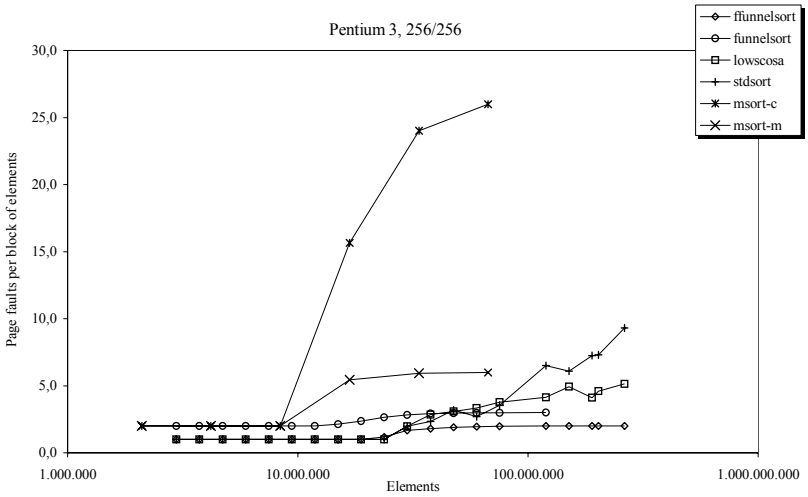


Chart C-65. Page faults sorting uniformly distributed pairs on Pentium 3.

The picture is identical on the two Pentium architectures. We can see that with very high accuracy, all in-place algorithms and the ffunnelsort incur N/B page faults and all merge-based algorithms incur $2N/B$ page faults. When input cannot fit in RAM, we also see that funnelsort incurs exactly $3N/B$ page faults and the ffunnelsort incurs $2N/B$. This is exactly as expected, since the cost of writing of the sorted result is not included in this measure.

The LOWSCOSA settles in at about 4-5 complete scans, while std::sort again continues to incur an increasing number of page.

We see that the tiled mergesort incurs a lot more page faults than any of the other algorithm does. Again, this is due to the many passes over the input. Multi mergesort incurs up to $6N/B$ page faults. It forms runs by scanning the entire input and generating the runs in an equal sized array. These runs are then scanned and the output written back in the original array. This accounts for only $4N/B$ page faults. We cannot account for the remaining $2N/B$. We suspect it is an unneeded pass, like the copying of all elements from the output to the input, in the middle of the algorithm.

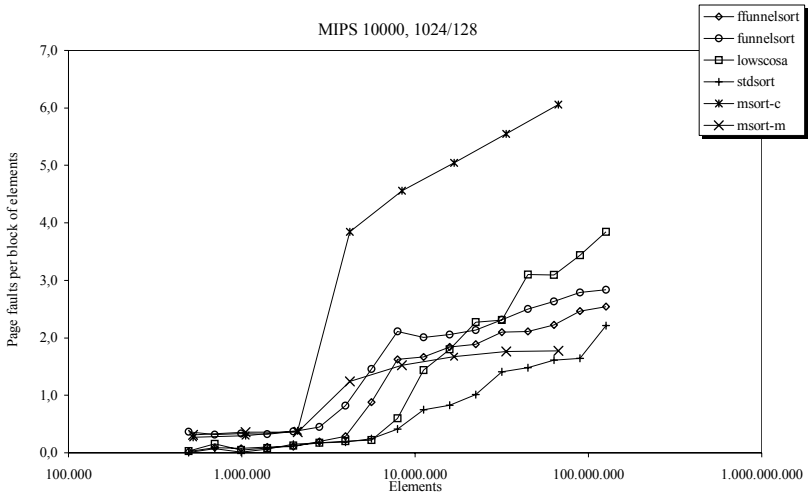


Chart C-66. Page faults sorting uniformly distributed pairs on MIPS 10000.

The numbers are a lot different on the MIPS computer. Here we are unable to explain the page fault count by the number of passes over the data, the algorithms make.

IRIX supports using several different page sizes. The getpagesize system call reveals that the page size of this particular system is 8KB. The values in the chart are based on this value. However, getpagesize may return a lower number to indicate that the allocation granularity is 8KB and not necessarily the actual page size used by the operating system, which may then be up to 64KB.

Another explanation is that this is the effect of the operating system prefetching pages. That is, if it can detect that a process is streaming through data, it may read in 8 or 16 pages per page fault. According the manual pages, the number of page faults reported by the getrusage system call is the number of memory operations that has caused an I/O. If indeed the operating system chooses to prefetch pages, this number will be significantly lower. An indication that this may be the case is that as the datasets get very large, the number of page faults caused by funnelsort and ffunnelsort comes closer to the expected values. This may then be because funnelsort is accessing so many streams at once that the operating system is unable to detect a streaming behavior and thus opts to not prefetch pages.

The next chart shows the number of cache misses incurred on the MIPS computer.

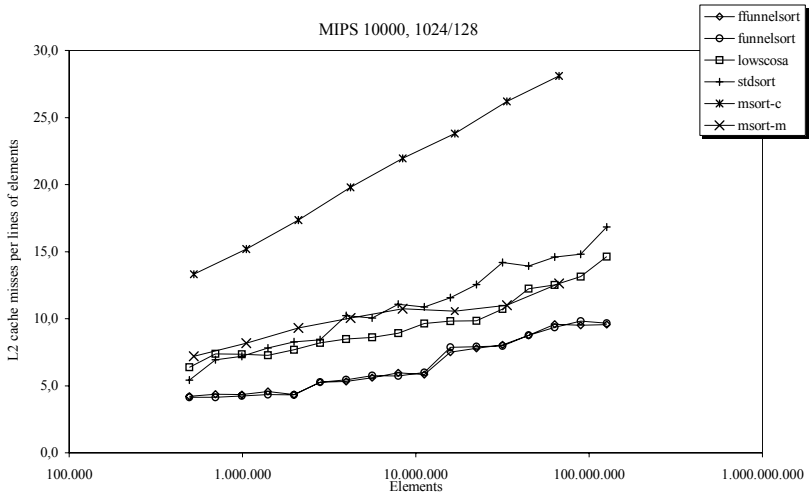


Chart C-67. Cache misses sorting uniformly distributed pairs on MIPS 10000.

This is indeed an interesting result. It clearly shows that funnelsort is able to maintain a very high degree of cache utilization, even on lower level caches, where the assumptions of the ideal cache model, such as full associativity and optimal replacement, most certainly does not hold. Even the LOWSCOSA incurs fewer cache misses than the other algorithms.

It is interesting to see that even for such small datasets as less than one million pairs, the high number of passes done by tiled mergesort causes a significant number of cache misses. It is also interesting to see that multi mergesort is not able to keep up with funnelsort. This is most likely due to a high number of conflict misses.

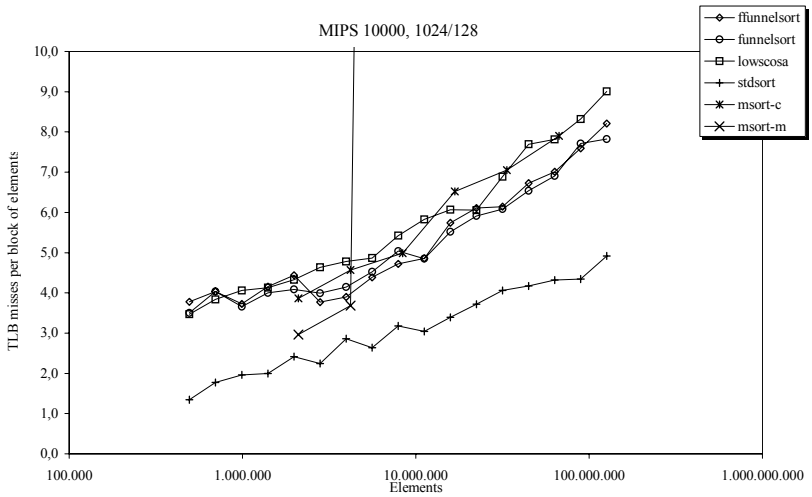


Chart C-68. TLB misses sorting uniformly distributed pairs on MIPS 10000.

The std::sort incurs the fewest TLB misses. The reason funnelsort is not as dominating on the TLB level of the hierarchy is likely because the TLB is not as tall as

the L2 cache or RAM. With only 64 TLB entries, the output of at most a 64-funnel can be merged with B^{-1} TLB misses per element, before another sub-funnel is loaded. This is likely significantly less than what is possible on the other levels of the hierarchy, where the order of the j-funnel is likely bounded by the capacity of the cache, rather than the number of blocks it contains.

The multi mergesort incurs more that 100 times more TLB misses and goes off the scale, when it sorts 2^{22} or more elements. 2^{22} elements correspond exactly to 64 cache-loads, which again corresponds to one cache-load per TLB entry. Merging more streams than there are TLB entries will cause thrashing.

It is interesting to note that neither the L2 cache misses nor the TLB misses is reflected significantly in the wall clock time, except perhaps for the extreme cases of the multi and tiled mergesorts.

6.2.2 Integer Keys

For the remainder of this chapter, we will look for changes in performance characteristics when the algorithms are applied to different data types. Unfortunately, due to time constraints not all algorithms and architectures could participate in the remaining the benchmarks. Let us first look at the performance when sorting only integers.

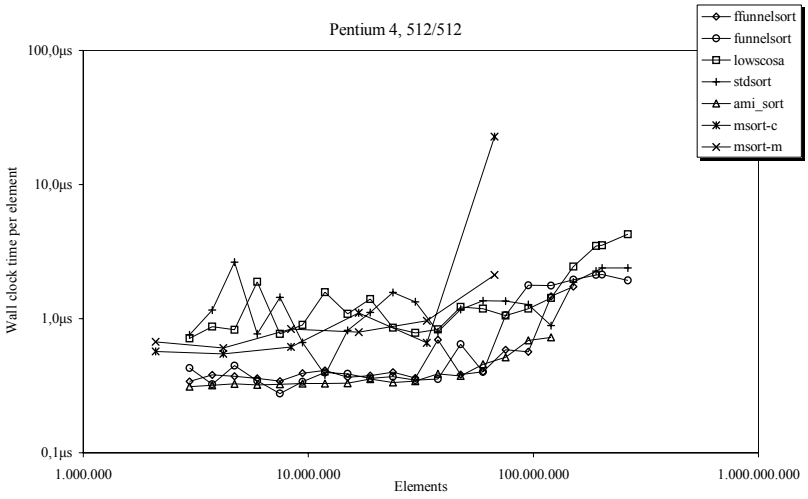


Chart C-69. Wall clock time sorting uniformly distributed integers on Pentium 4.

On the Pentium 4, the ami_sort still dominates, albeit not by as much as when sorting pairs. We see some large fluctuations in the input-sensitive algorithms based on partitioning. This could be due to unlucky pivot selection; however, it may also simply be caused by external influences. Funnelsort is very competitive and std::sort fluctuates a lot. Other than that there is not much new.

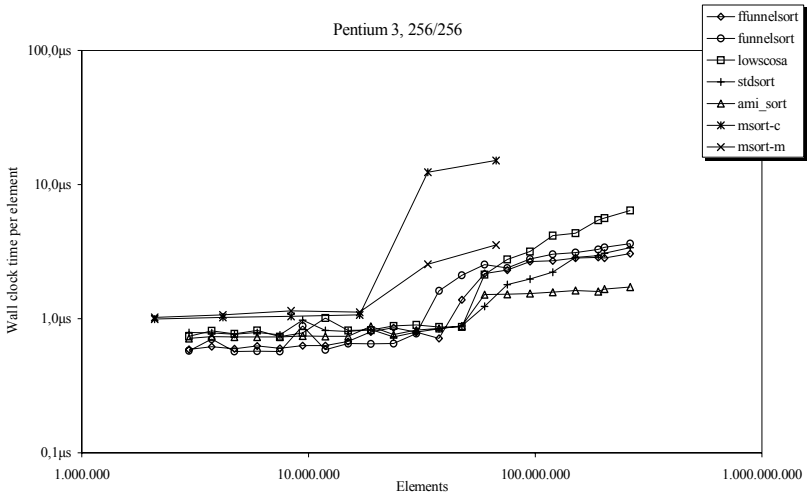


Chart C-70. Wall clock time sorting uniformly distributed integers on Pentium 3.

On the Pentium 3, the picture is largely unchanged from when sorting pairs.

6.2.3 Records

In this section, we sort uniformly distributed records of size 100 bytes each. Using such large elements reduces the number of elements within a given part of memory, thus fewer comparisons are done to sort the elements within the same amount of space. This should in turn downplay the cost of a comparison compared to cache effects.

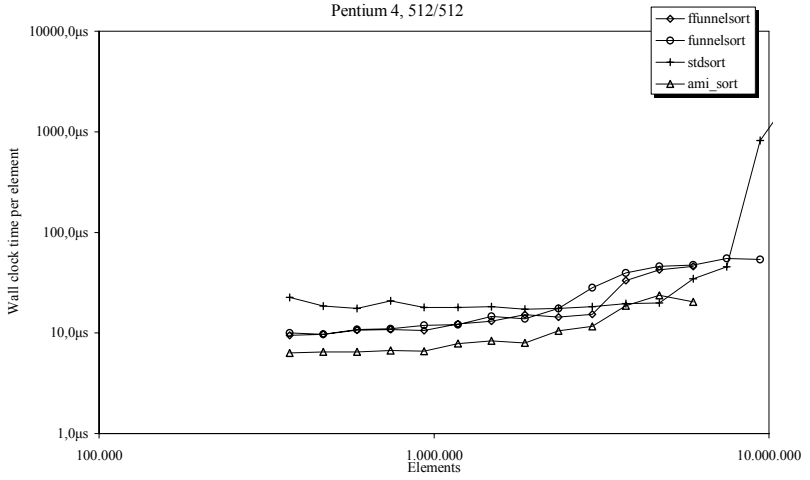


Chart C-73. Wall clock time sorting uniformly distributed records on Pentium 4.

Again, we see ami_sort dominate on the Pentium 4. Close second and third are the funnelsorts and slowest is std::sort, exactly as when sorting pairs. This leads us to believe that cache effects rather than just comparisons are also very important when sorting small elements.

std::sort exhibits a dramatic jump in running time when input can no longer fit in RAM. We can see from the page fault count that it is accompanied by an increase in total incurred page faults:

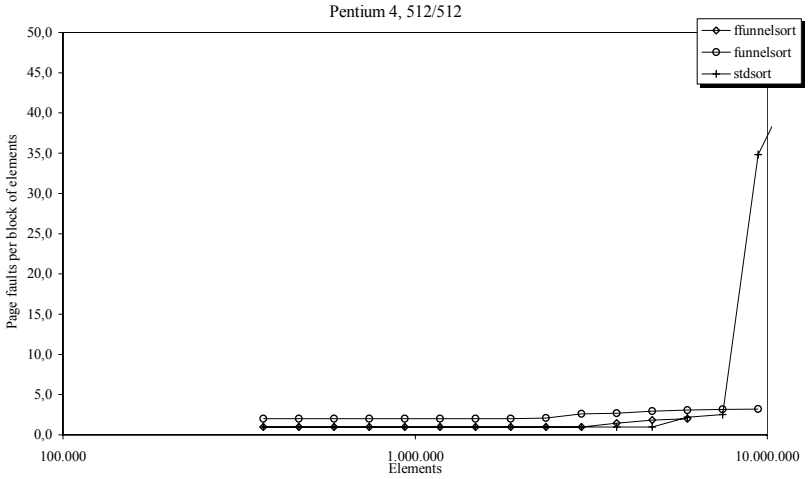


Chart C-75. Page faults sorting uniformly distributed records on Pentium 4.

The reason for this jump is not entirely clear.

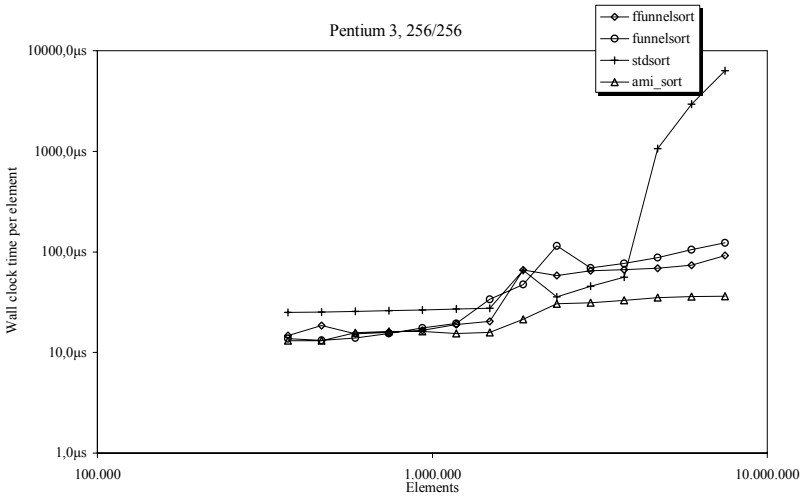


Chart C-74. Wall clock time sorting uniformly distributed records on Pentium 3.

On the Pentium 3, the ami_sort now performs on par with the funnelsorts. std::sort is significantly slower and exhibits the same increase in running time as seen on the Pentium 4.

6.3 Special Cases

In this section, we investigate the performance of the algorithms when sorting special distributions of elements. std::sort and the LOWSCOSA are both input sensitive and may thus react differently to different distributions. Even though merge-based sorting algorithms (as implemented here) are not considered input-insensitive, certain distributions may make branches in the code more predictable, and thus influence merge-based sorting algorithms as well.

6.3.1 Almost Sorted

The first distribution we will look at mimics an almost sorted dataset that needs to be completely sorted. A sample distribution of 1024 elements can be seen in Figure 6-3, where there is six buckets of elements in a given range, such that all elements in one bucket is smaller than any element in the next. In general, there will be $\ln(n)$ buckets in a distribution of n elements.

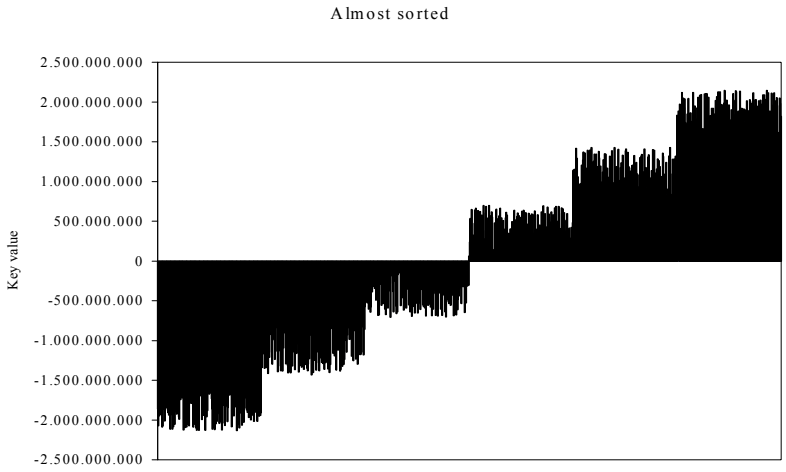


Figure 6-3. Almost sorted key distribution.

The results are as follows:

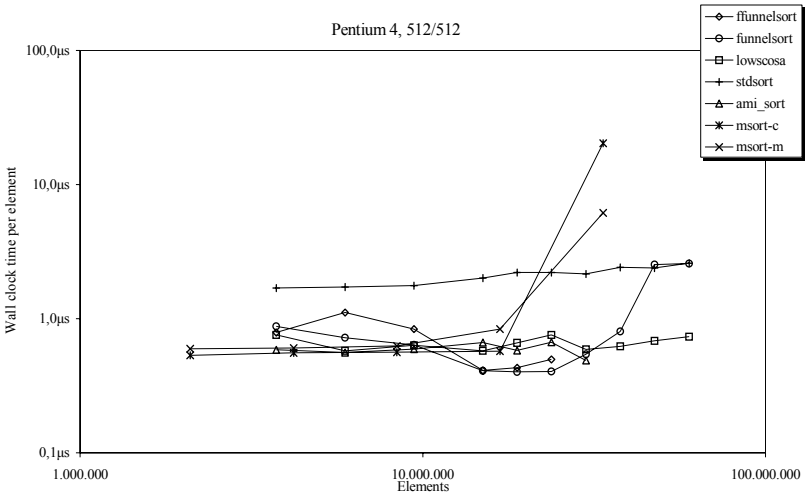


Chart C-77. Wall clock time sorting almost sorted pairs on Pentium 4.

Comparing to Chart C-61, we can see that std::sort performs slightly worse on almost sorted data. This is indeed unexpected and cannot immediately be explained. We would expect a partitioning operation on almost sorted data not to swap many elements, in turn not to cause the referenced bit to be set and thus save many expensive writes to disk. Other than that, no significant changes can be seen.

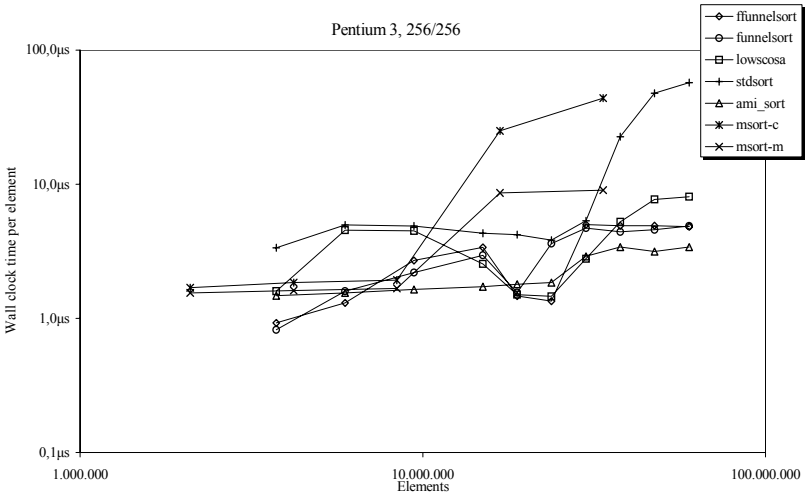


Chart C-78. Wall clock time sorting almost sorted pairs on Pentium 3.

Comparing to Chart C-62, we can again see an increase in the running time of std::sort. We can also see that the funnelsorts increase their per element running time as the size of the dataset increases up to about 15,000,000 elements.

6.3.2 Few Distinct Elements

The other distribution we will look at contains many elements but only few distinct ones. A sample distribution of 1,024 elements with six distinct keys can be seen in Figure 6-4. In general, there will be $\ln(n)$ distinct keys in a distribution of n elements.

6.3.2 Few Distinct Elements

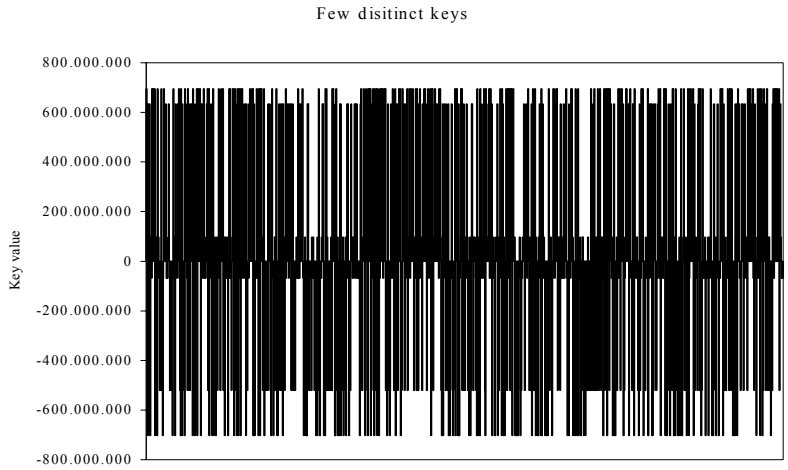


Figure 6-4. Few distinct keys distribution.

The results are as follows:

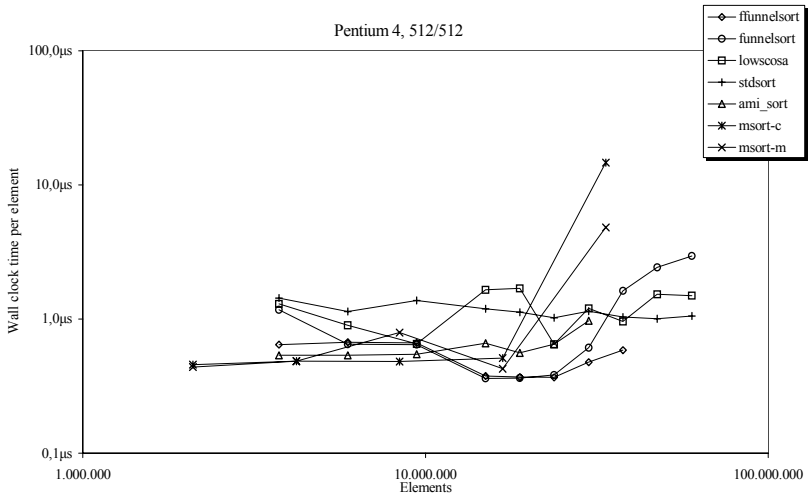


Chart C-81. Wall clock time sorting few distinct pairs on Pentium 4.

This result looks like the result for uniform distribution, with the running time per element of the funnelsorts now being more constant.

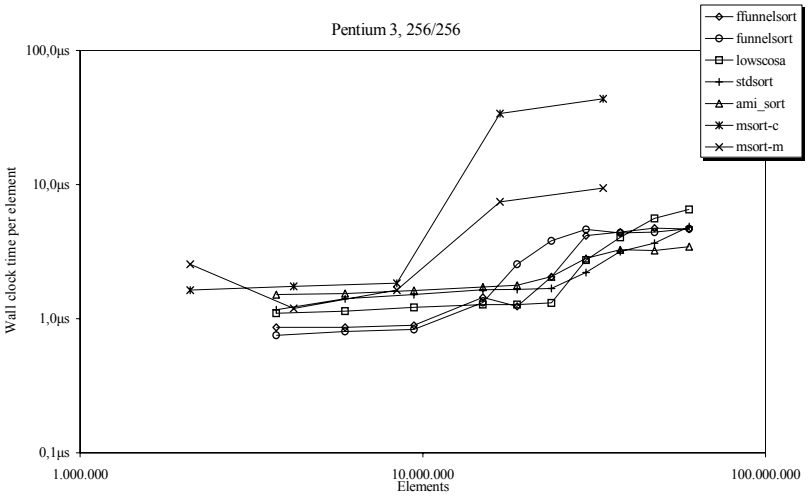


Chart C-82. Wall clock time sorting few distinct pairs on Pentium 3.

As does this to a high degree. If anything, the funnelsorts appear to perform slightly better, relative to std::sort, than when sorting uniformly distributed elements. Had std::sort been using a Dutch flag partitioning, it would have probably been faster.

Chapter 7

Conclusion

In this thesis, we set out to produce a high performing cache-oblivious sorting algorithm in part to clarify the feasibility of cache-oblivious algorithms in the context of sorting.

This feasibility is not a given. Cache-oblivious algorithms are proven optimal in the memory hierarchy using assumptions that appear unrealistic. Not before an implementation of the algorithms has been created and thoroughly analyzed experimentally, will it become clear whether these assumptions are indeed unrealistic, or cover for aspects of the realities of modern day hardware that are important for achieving high performance. Furthermore, optimal cache-oblivious sorting algorithms are more complex and require more instructions to be executed per element sorted. Thus, it is unclear whether this increased complexity will cause the algorithm to perform badly despite any improvement in cache utilization. Again, only an experimental analysis of the algorithms will provide a clear answer.

Realizing that space consumption is of great importance when working with large datasets, we have developed a novel low-order working space cache-oblivious sorting algorithm, LOWSCOSA. It has optimal complexity and uses sub-linear working space to sort elements, keeping them in the array in which they are stored.

We have provided an implementation of the cache-oblivious sorting algorithm funnelsort and an implementation of the LOWSCOSA.

Using a detailed knowledge of the inner workings of both compilers, modern processors and modern memory systems, we build an understanding of what ingredients are needed in a high performance cache-oblivious sorting algorithm. We have used a process of thorough experimentation to determine exactly which ingredients improve performance in practice. Through this process, many good ideas were tried out in practice. However, only very few proved able to yield improvements in performance. Though it may seem unfortunate that we were unable to improve performance, using approaches that are more sophisticated, it is indeed not. By showing that performance does not improve significantly when using complex solutions, we have also shown that the simplest implementation of these algorithms will likely be as fast as or even faster than solutions that are more complex. That fast implementations of algorithms can be created using only simple techniques, is important for the spread and acceptance of the algorithm.

In particular, we have shown both theoretically and in practice that perhaps the most complex aspect of the algorithms, namely the controlled layout, is of minimal importance for performance.

This thesis breaks new ground by proving that implementations of cache-oblivious sorting algorithms can have performance comparable or even superior to popular alternatives. We have provided evidence that the assumptions, such as full associativity, made to argue optimal utilization of cache, is no hindrance to achieving high cache utilization. Indeed, we have shown that cache-oblivious algorithms can exploit the caches even better than algorithms tuned for the cache and in turn achieve higher performance.

We have also shown that these results are consistent through several different types of input and on several radically different hardware architectures.

However, our cache-oblivious algorithms are not able to outperform algorithms designed and implemented specifically to be efficient in handling disk accesses.

7.1 Further Improvements of the Implementation

We believe the performance of our implementations can be further improved. Our implementation is build as a set of clear-cut abstractions. This was done to ensure modularity, enabling us to try out different pieces of code in the same contexts. However, the compiler may not be able to see through all of these abstractions, making it hard for it to generate optimal code.

Now that we have determined what pieces of code yield high performance, this structure of abstractions is no longer needed. We believe, implementing the algorithms from scratch using these pieces of code but without the abstractions, will yield a higher performing implementation.

We believe that to achieve the performance levels of algorithms designed and implemented for efficient handling of disk accesses, either the I/O of the operating system need to be reworked, or cache-oblivious algorithms need to begin utilizing the same techniques. These techniques include double buffering and prefetching. Using such techniques would likely also result in a faster implementation.

7.2 Further Work

When using constant sized samples for finding medians for use in partitions, we risk making uneven partitions. For performance reasons, we cannot afford to find the exact median in the LOWSCOSA. This in turn means that in practice, we run the risk of quadratic complexity. To make the LOWSCOSA more compelling, we need to find a way to guarantee worst-case optimality in practice.

Depending on the choice of parameters used in the algorithm, the LOWSCOSA still consumes a significant amount of additional memory. We would like to see this amount reduced to a logarithmic term or perhaps constant.

The LOWSCOSA does not have competitive performance in practice for large datasets. It incurs $6N/B$ memory transfers, where only $4N/B$ is needed. We would like to see a variant of the LOWSCOSA that also had a competitive performance on large datasets.

Still only very little work has been done in the experimental study of cache-oblivious algorithms. A lot more ground needs to be covered. It is the hope of the author that this thesis will help pave the way for more work to be done in this area.

Appendix A

Electronic Resources

Accompanying this thesis is a CDROM. The CDROM contains an electronic version of the thesis, the complete source code, and all benchmark results.

The contents of the CDROM are also available online, possibly in improved versions, so please point your browser to

http://www.brics.dk/~kv/thesis/

for latest editions.

A.1 The Thesis

This thesis is available as a Portable Document Format file located at the root of the CDROM. The source of the thesis is available in the /doc directory.

A.2 Source Code

The full source code is available in the /src directory. The Sort subdirectory contains the headers needed to use the sorting algorithms.

The /bin directory contains selected compiled binaries as well as makefiles to allow others to compile the code. Some aspects of the makefiles, such as compiler paths and include paths, may need to be adapted to individual installations.

A.3 Benchmark results

The results of all benchmarks are included in the /doc/charts directory.

Appendix B

Test Equipment

For reference, we provide the details of the equipment used in the benchmarks of Chapters 5 and 6.

B.1 Computers

We use three different computers for our tests. They are categorized as a Pentium 4, a Pentium 3 and a MIPS 10000 computer. Their specifications are as follows, according to [HSU+01], [MPS02], and vendor web sites:

Processor type	Pentium 4	Pentium 3	MIPS 10000
Workstation	Dell PC	Delta PC	SGI Octane
Operating system	GNU/Linux Kernel version 2.4.18	GNU/Linux Kernel version 2.4.18	IRIX version 6.5
Clock rate	2400 MHz	800 MHz	175 MHz
Address space	32 bit	32 bit	64 bit
Integer pipeline stages	20	12	6
L1 data cache size	8 KB	16 KB	32 KB
L1 line size	128 Bytes	32 Bytes	32 Bytes
L1 associativity	4 way	4 way	2 way
L2 cache size	512 KB	256 KB	1024 KB
L2 line size	128 bytes	32 bytes	32 bytes
L2 associativity	8 way	4 way	2 way
TLB entries	128	64	64
TLB associativity	Full	4 way	Full
TLB miss handler	Hardware	Hardware	Software
Main memory	512 MB	256 MB	128 MB

B.2 Compilers

The following compilers were used to build the executables available as described in Appendix A and used in the tests of Chapters 5 and 6

- *GNU Compiler Collection* version 3.1.1. Common compiler flags:

```
-DNDEBUG -O6 -fomit-frame-pointer -funroll-loops -fthread-jumps -ansi
-Wall -Winline -pedantic
```

- *Intel C++ Compiler* version 7.0. Common compiler flags:

```
-D NDEBUG -O3 -IPA -Ofast=ip30 -LANG:std -ansi -64 -mips4 -r10000
```

- *MIPS Pro C++ Compiler* version 7.3.1. Compiler flags:

```
-DNDEBUG -O3 -rcd -ipo -unroll -vec -w1
```

- *Microsoft Visual C++ Compiler* version 13.1. Compiler flags:

```
/Ox /Og /Oi /Ot /Oy /G6 /GA /D "WIN32" /D "NDEBUG" /D "_CONSOLE" /D
"_MBCS" /GF /FD /EHsc /ML /arch:SSE /Za /Zc:forScope /Fo"Release/"
/Fd"Release/vc70.pdb" /W4 /nologo /c /Wp64 /Zi /TP /wd4290
```

For compiling Windows executables, the `WIN32` macro needs to be defined and when using PAPI, the `PAPI` macro is defined. When building Pentium 3 executables the flag `-march=pentium3` was used with the GCC and `-tpp6 -xiMK` with the Intel compiler. When building Pentium 4 executables, `-march=pentium4` and `-tpp7 -xiMKW` was used.

As discussed in Section 6.1.2, the TPIE library is not written in C++, so we had to use version 2.96 of the GCC to build the `ami_sort` executable.

Appendix C

Supplementary Benchmarks

This is the complete collection of benchmarks run in conjunction with this thesis. See Appendix A for how to obtain the numerical values and the source code for the programs that generated them.

C.1 Layout and Navigation

The results of the benchmarks described in Sections 5.3.2 and 5.3.3.

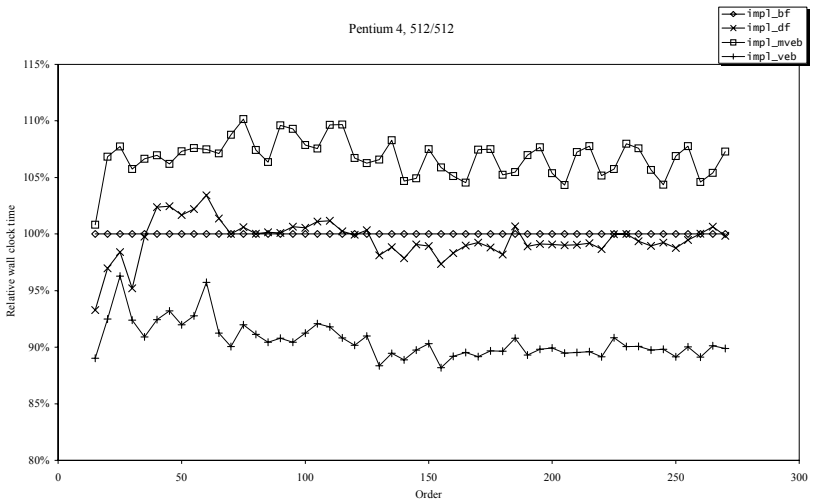


Chart C-1. Implicit layout on Pentium 4.

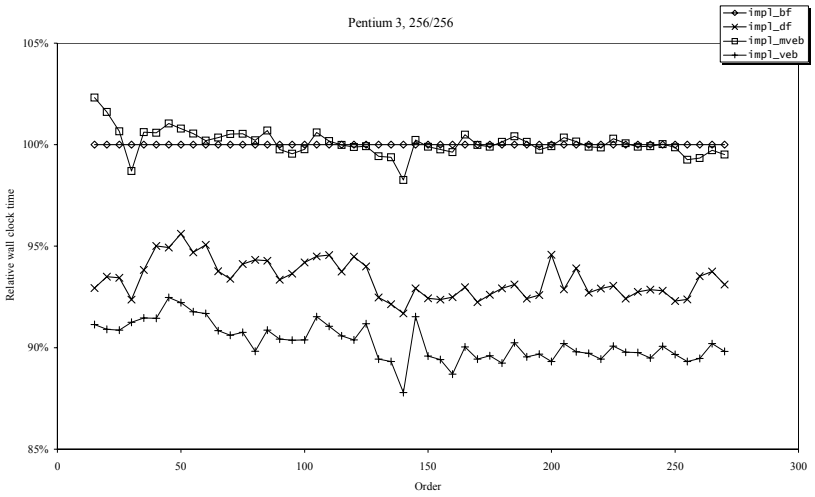


Chart C-2. Implicit layout on Pentium 3.

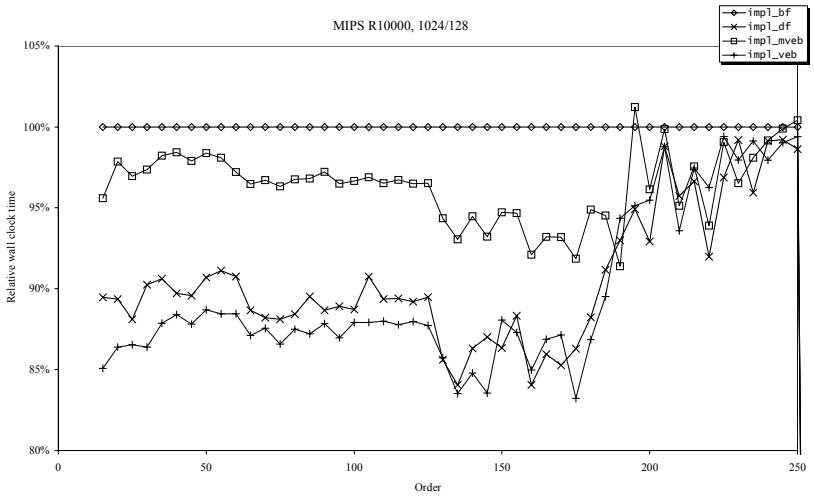


Chart C-3. Implicit layout on MIPS R10000.

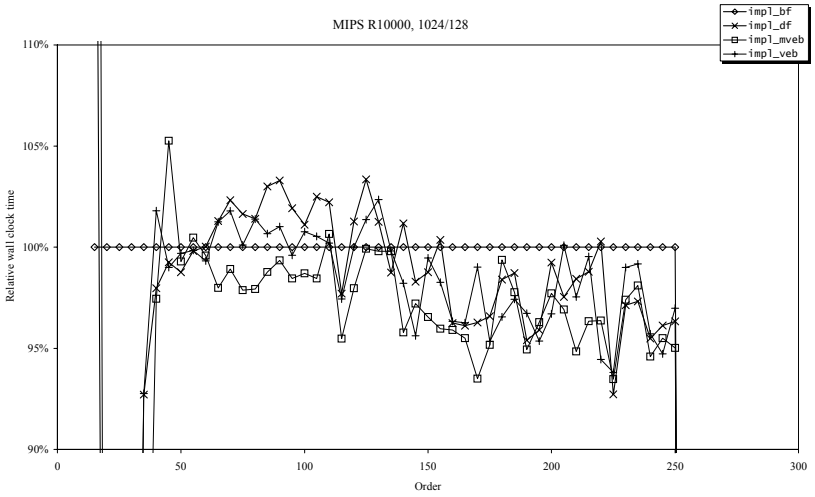


Chart C-4. Implicit layout on MIPS R10000, relative L2 cache misses.

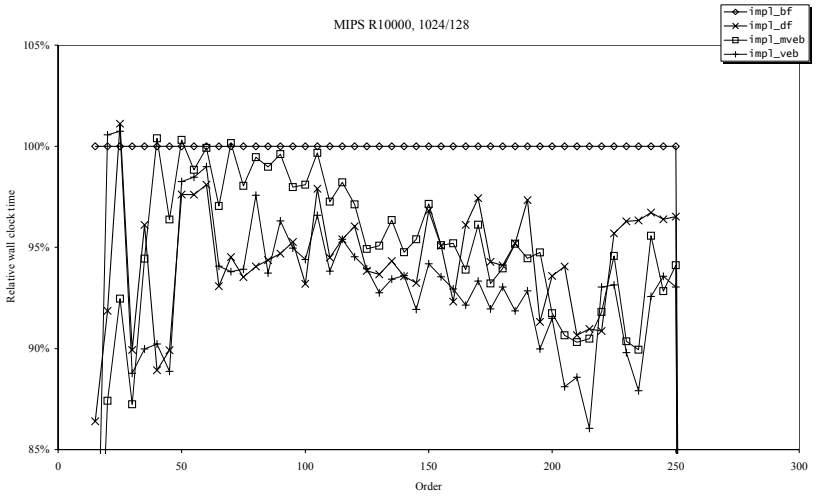


Chart C-5. Implicit layout on MIPS R10000, relative TLB misses.

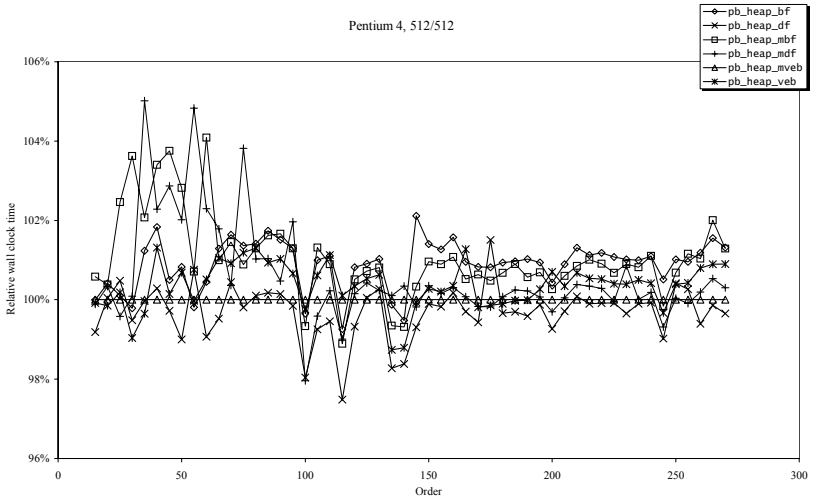


Chart C-6. Layout using std::allocator on Pentium 4.

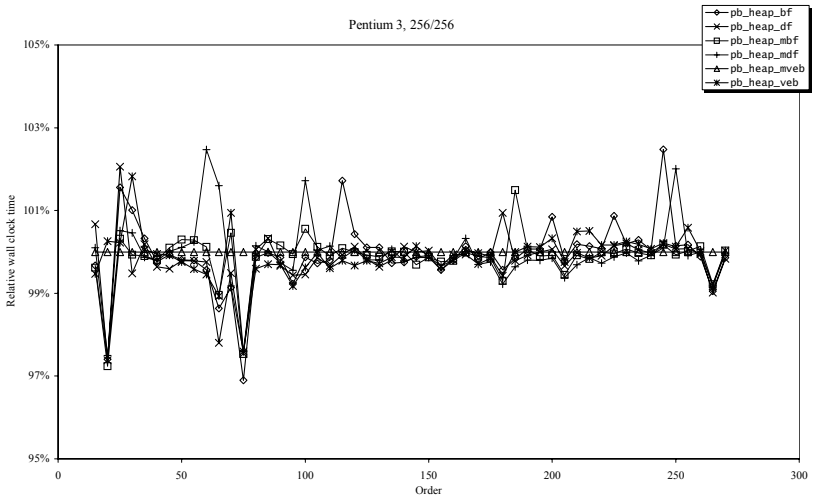


Chart C-7. Layout using std::allocator on Pentium 3.

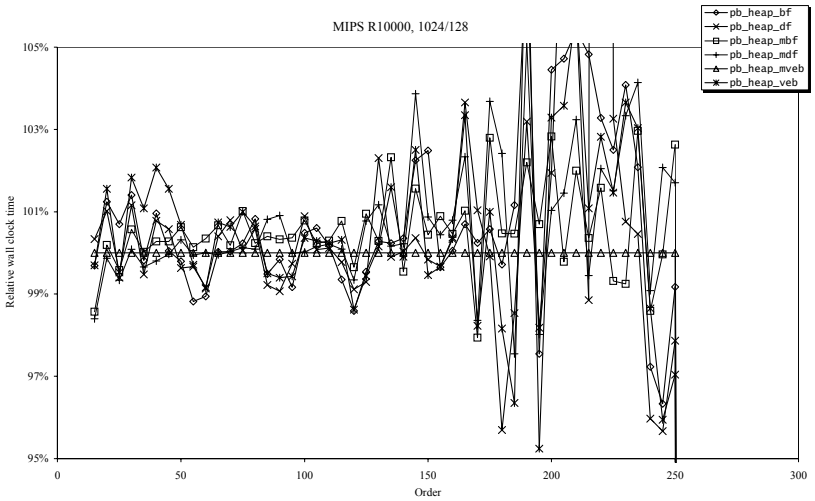


Chart C-8. Layout using std::allocator on MIPS 10000.

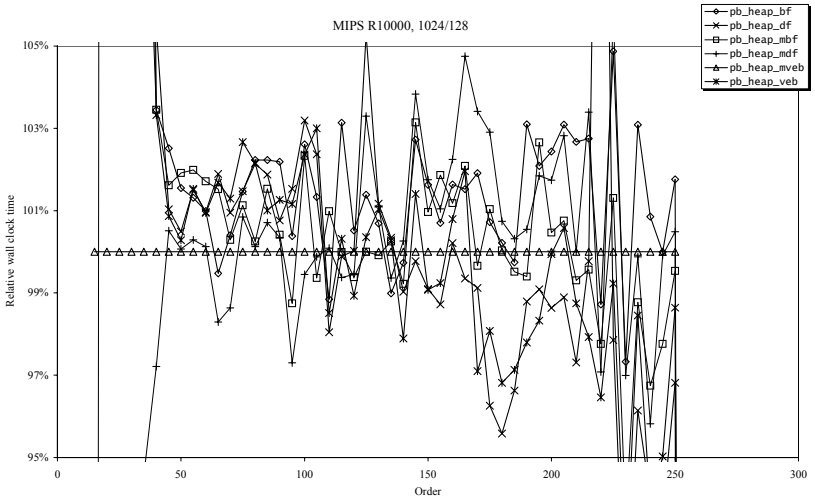


Chart C-9. Layout using std::allocator on MIPS 10000, L2 cache misses.

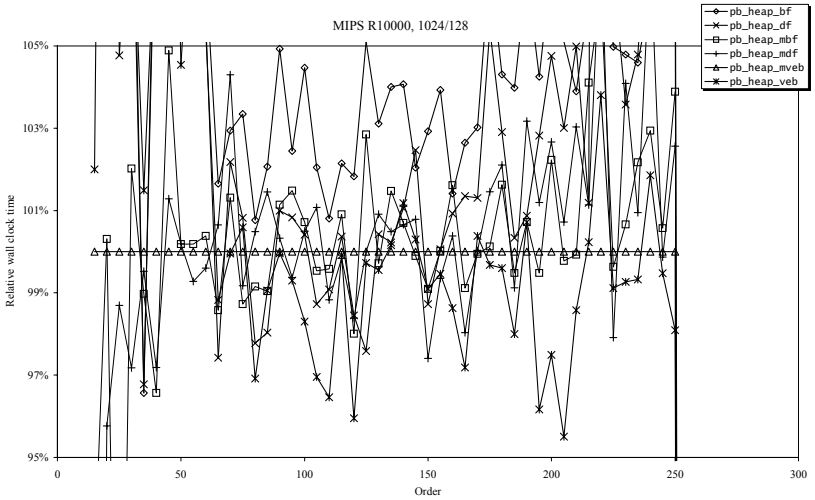


Chart C-10. Layout using std::allocator on MIPS 10000, TLB misses.

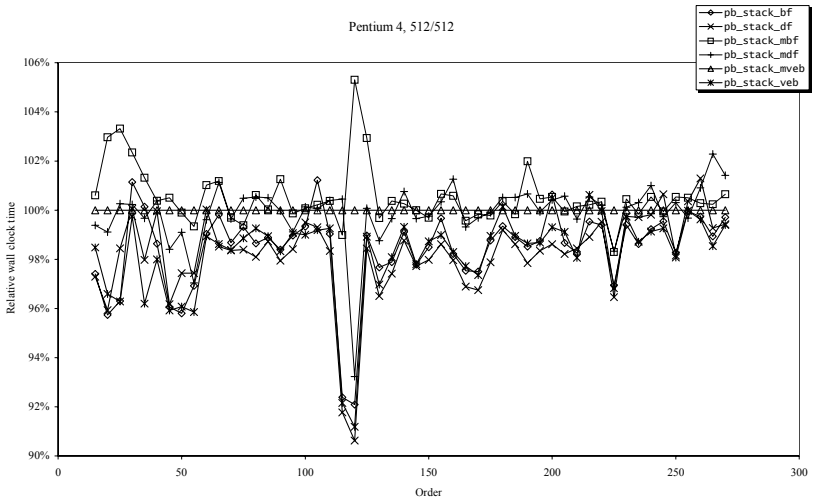


Chart C-11. Layout using stack_allocator on Pentium 4.

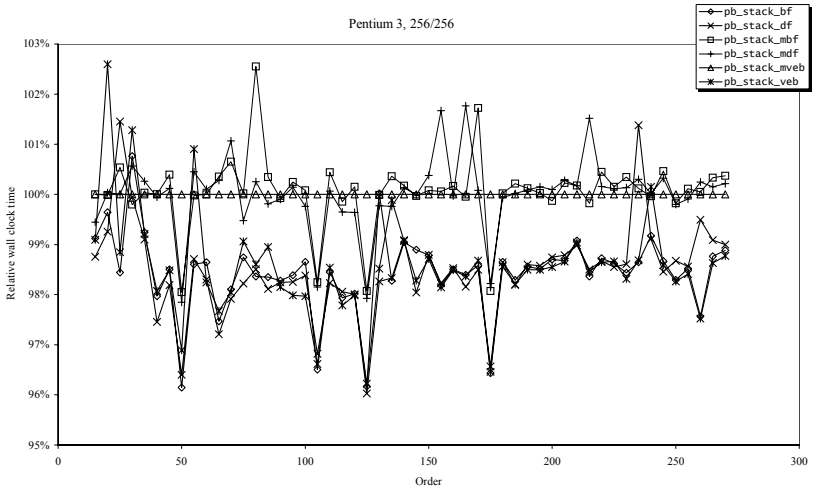


Chart C-12. Layout using stack_allocator on Pentium 3.

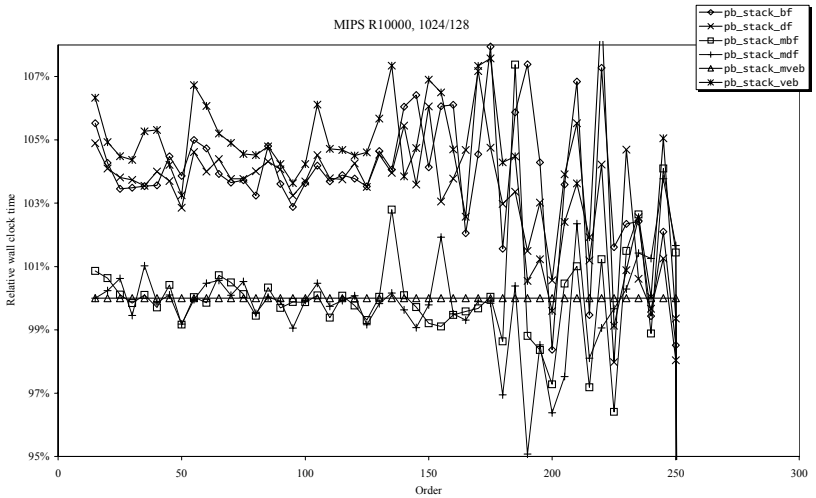


Chart C-13. Layout using stack_allocator on MIPS 10000.

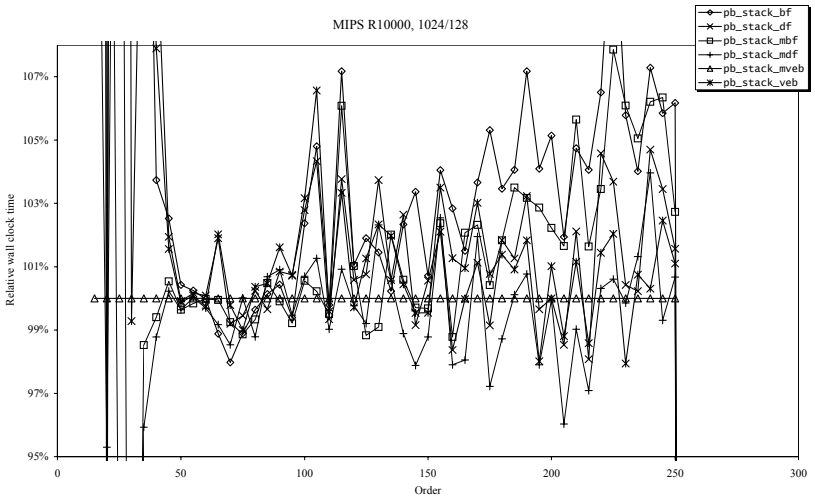


Chart C-14. Layout using stack_allocator on MIPS 10000, L2 cache miss.

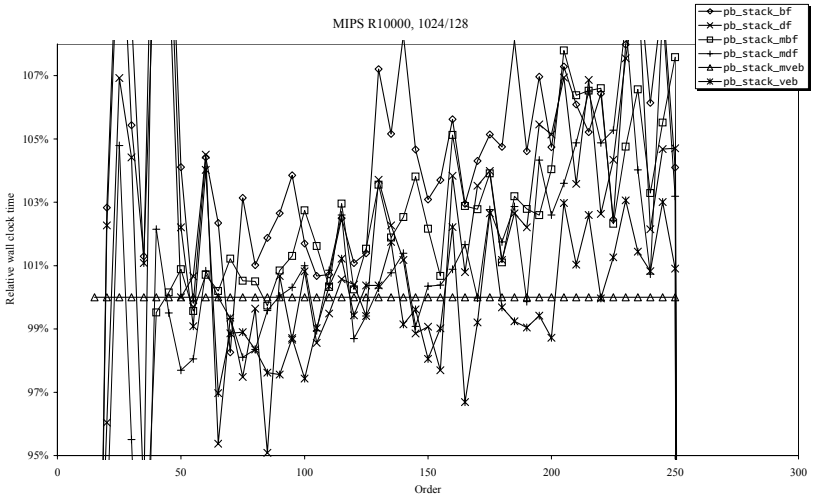


Chart C-15. Layout using stack_allocator on MIPS 10000, TLB miss.

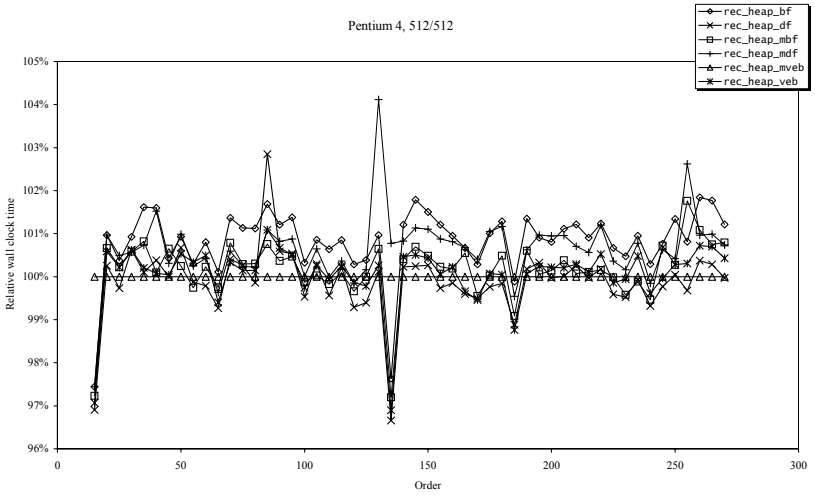


Chart C-16. Recursive fill, heap, Pentium 4.

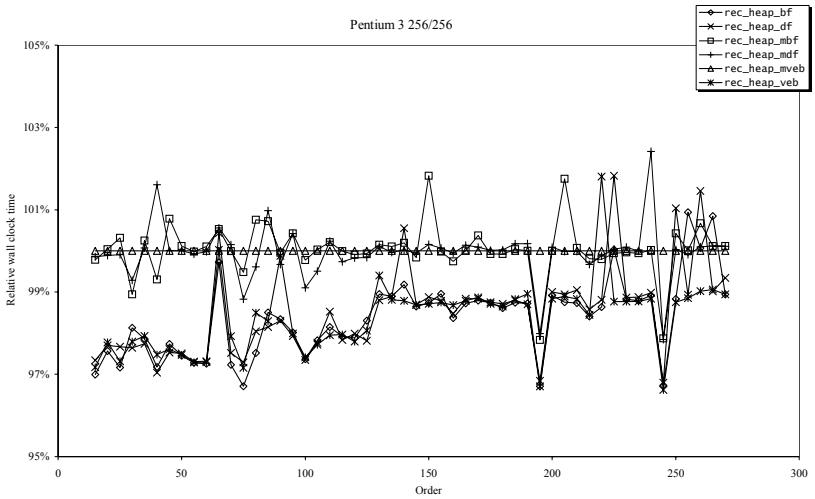


Chart C-17. Recursive fill, heap, Pentium 3.

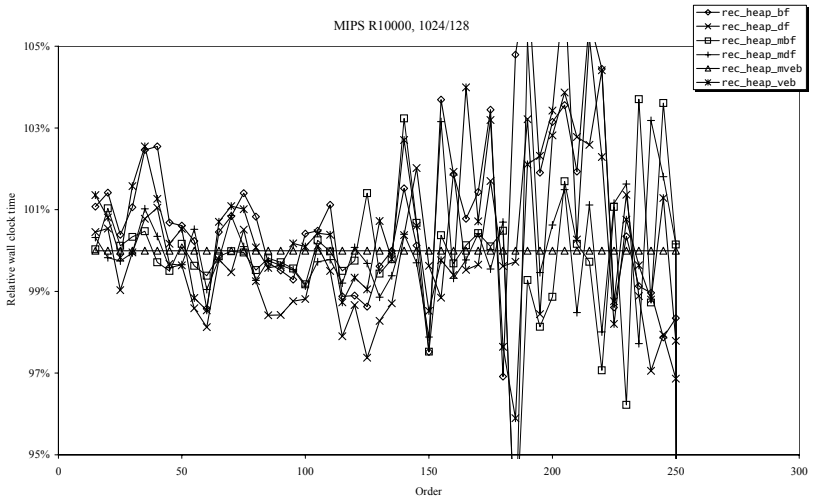


Chart C-18. Recursive fill, heap, MIPS 10000.

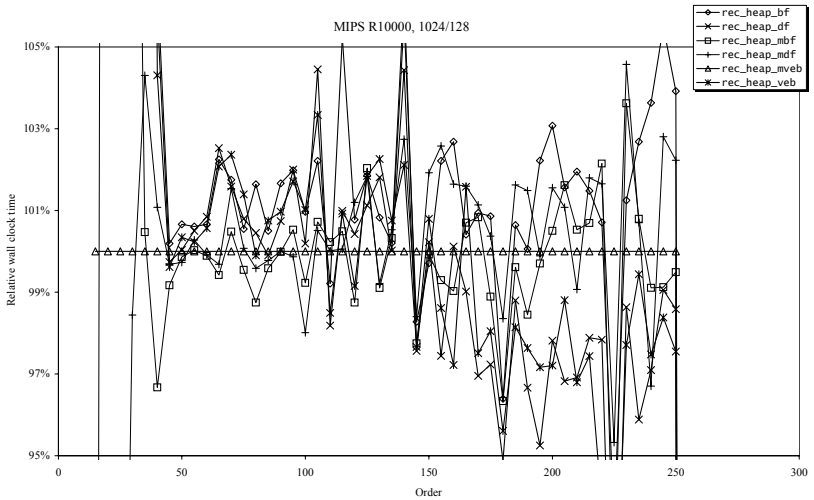


Chart C-19. Recursive fill, heap, MIPS 10000, L2 cache miss.

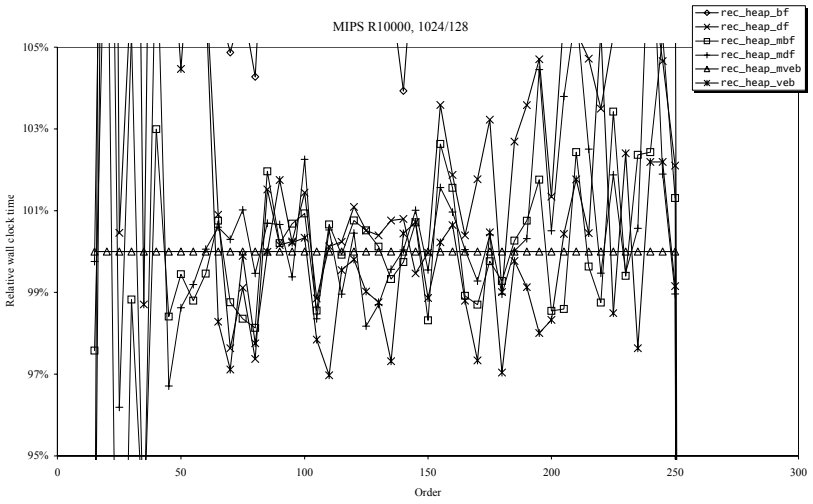


Chart C-20. Recursive fill, heap, MIPS 10000, TLB miss.

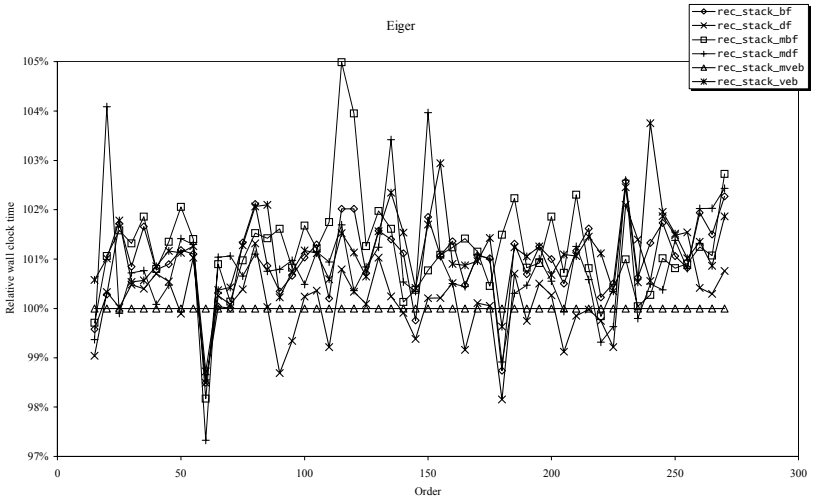


Chart C-21. Recursive fill, stack, Pentium 4.

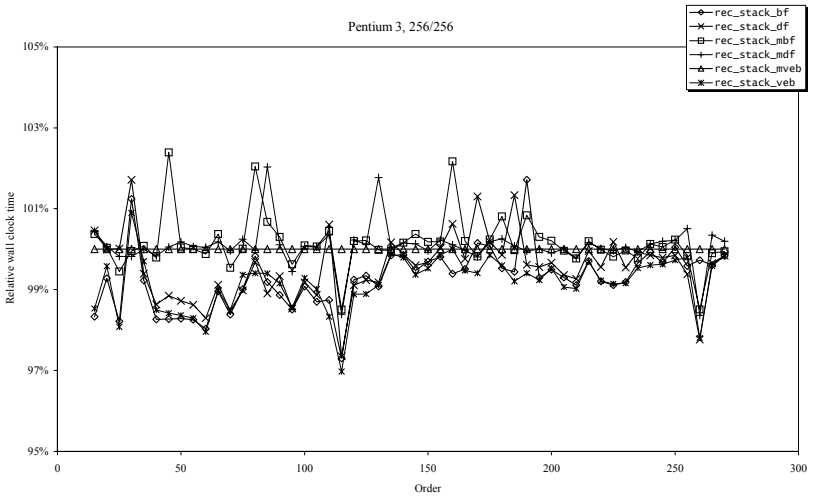


Chart C-22. Recursive fill, stack, Pentium 3.

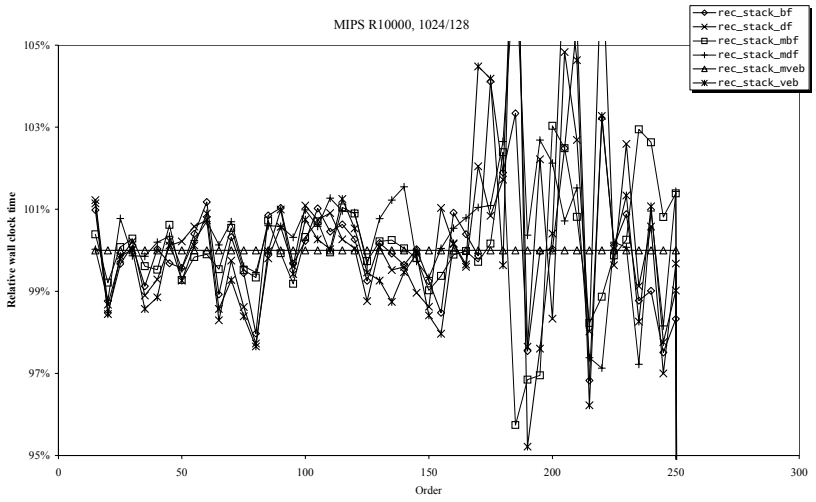


Chart C-23. Recursive fill, stack, MIPS 10000.

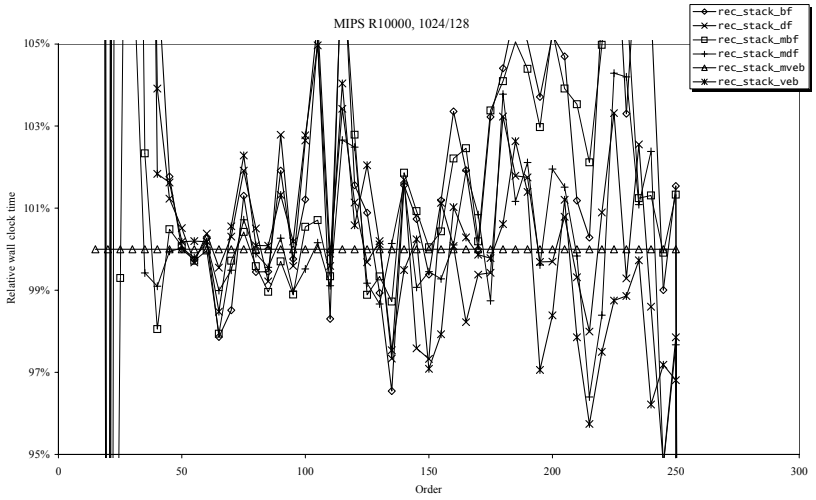


Chart C-24. Recursive fill, stack, MIPS 10000, L2 cache miss.

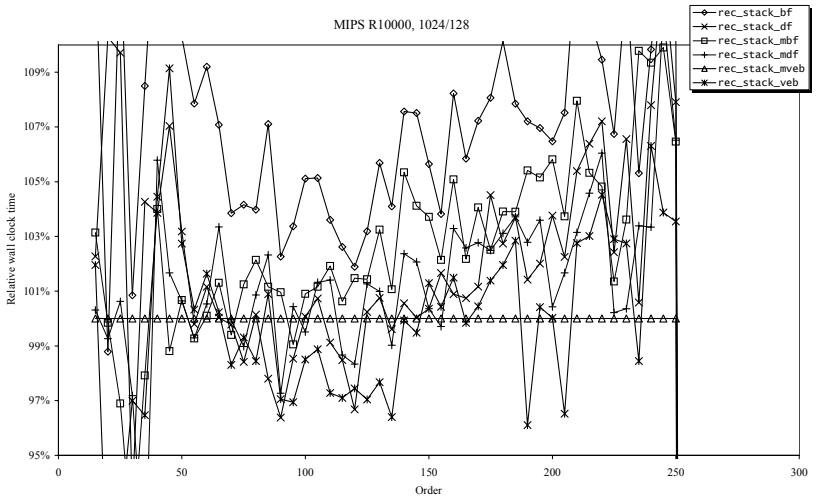


Chart C-25. Recursive fill, stack, MIPS 10000, TLB miss.

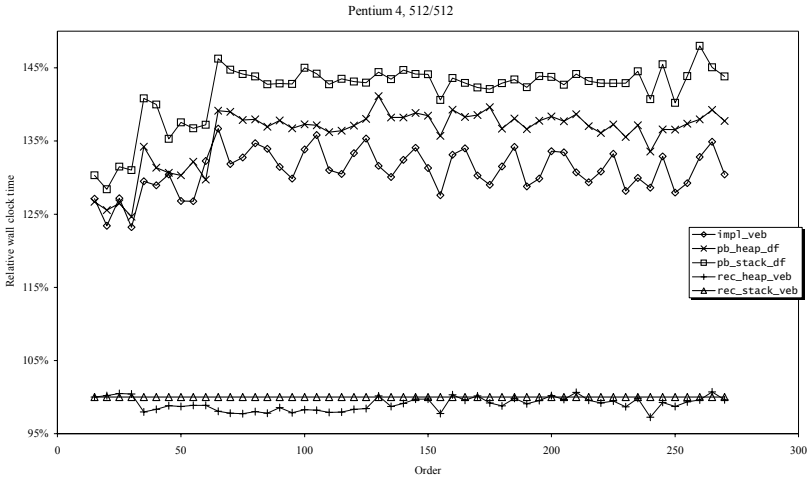


Chart C-26. Final, Pentium 4.

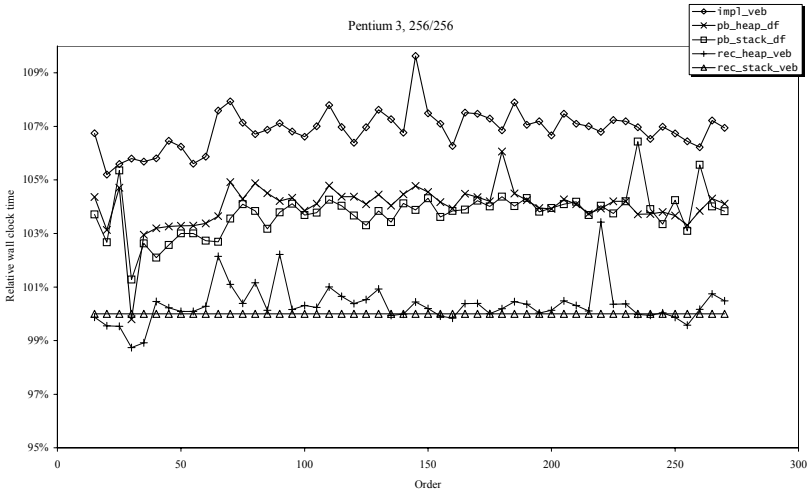


Chart C-27. Final, Pentium 3.

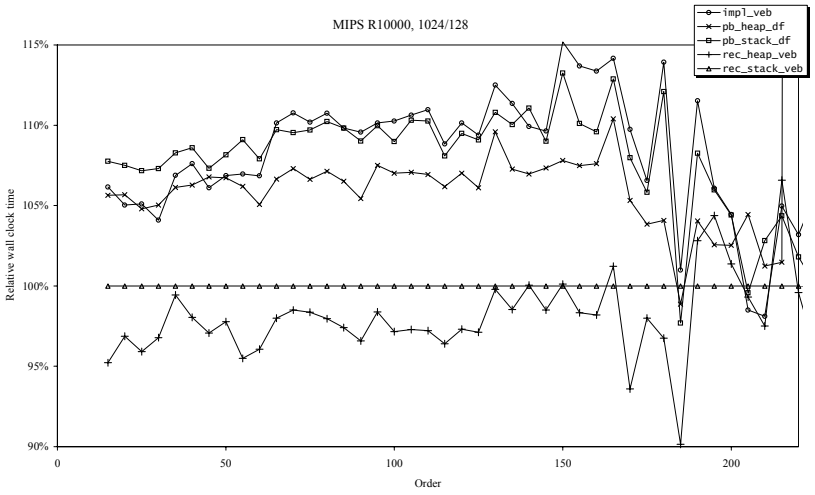


Chart C-28. Final, MIPS 10000.

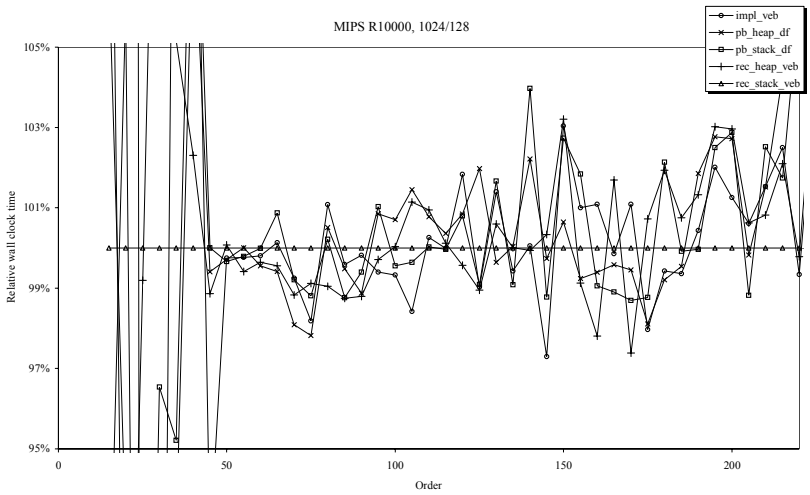


Chart C-29. Final, MIPS 10000 L2 cache misses.

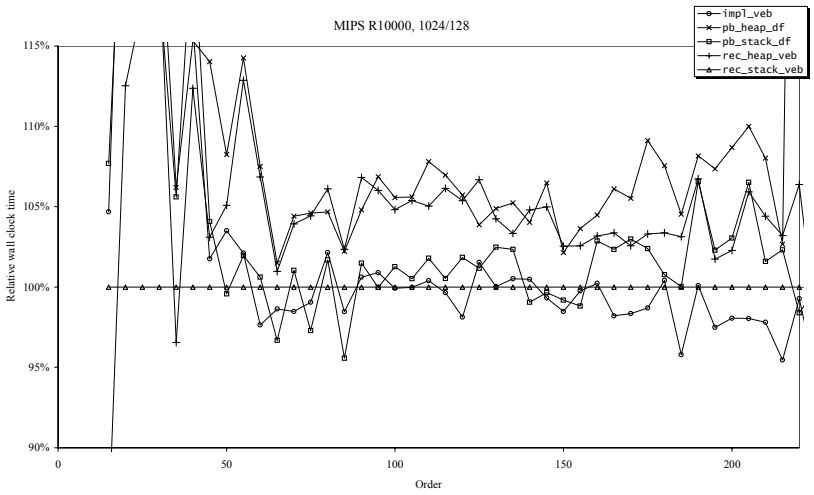


Chart C-30. Final, MIPS 10000 TLB miss.

C.2 Basic Merger

The results of the benchmarks described in Section 5.3.4.

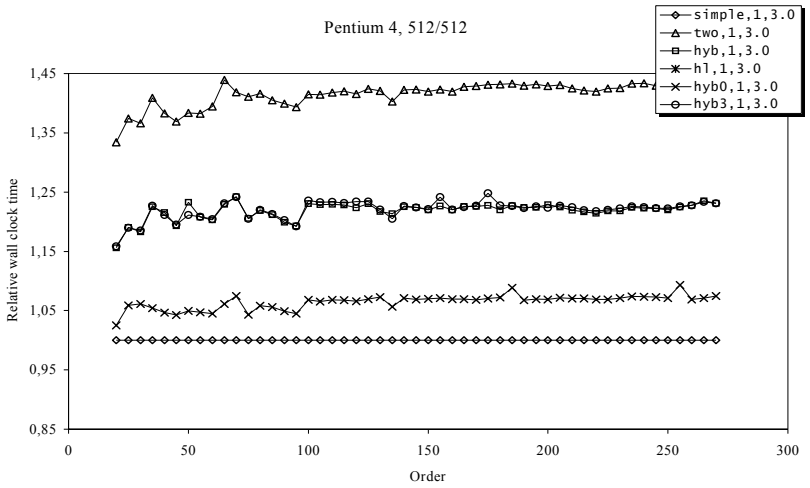


Chart C-31. Basic mergers, $(\alpha,d) = (1,3)$, Pentium 4.

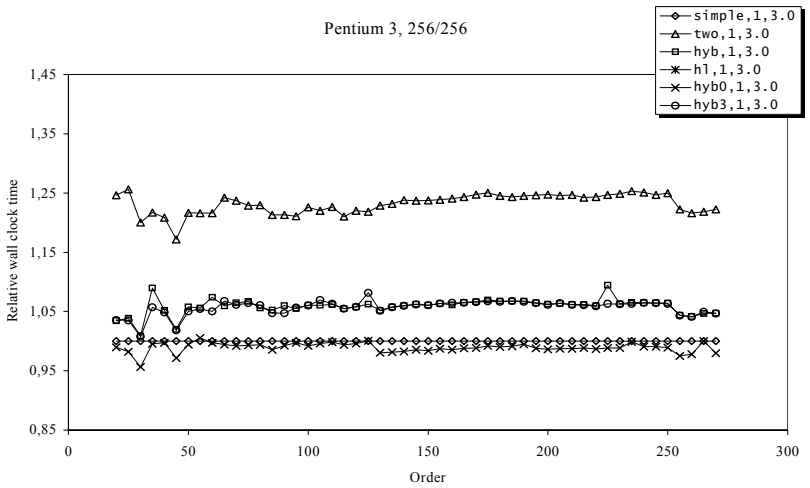


Chart C-32. Basic mergers, $(\alpha,d) = (1,3)$, Pentium 3.

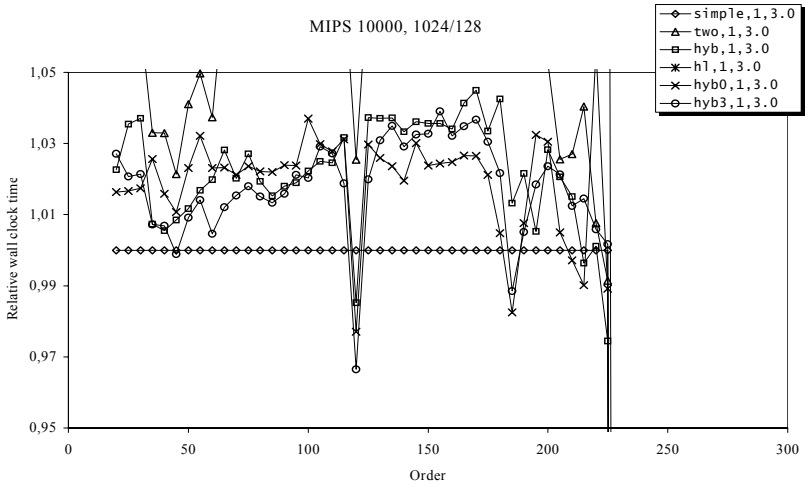


Chart C-33. Basic mergers, $(\alpha,d) = (1,3)$, MIPS 10000.

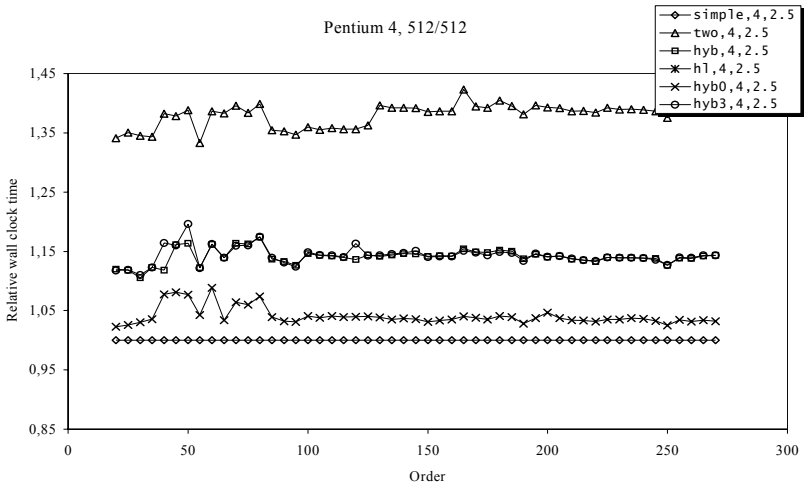


Chart C-34. Basic mergers, $(\alpha,d) = (4,2.5)$, Pentium 4.

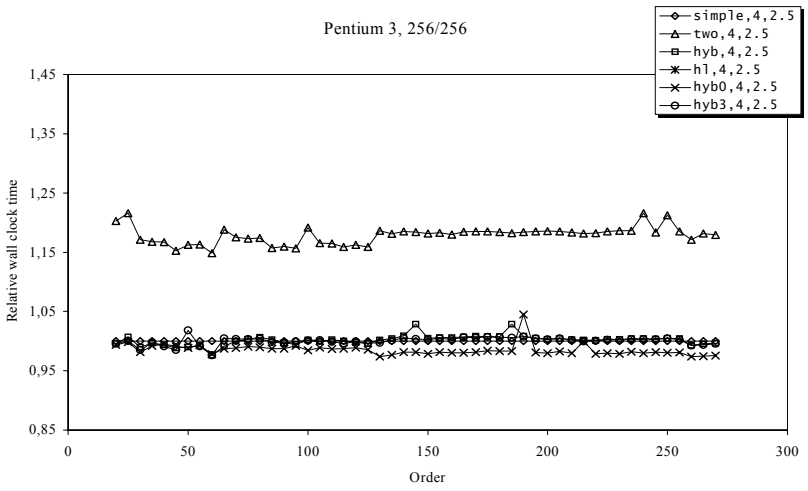


Chart C-35. Basic mergers, $(\alpha,d) = (4,2.5)$, Pentium 3.

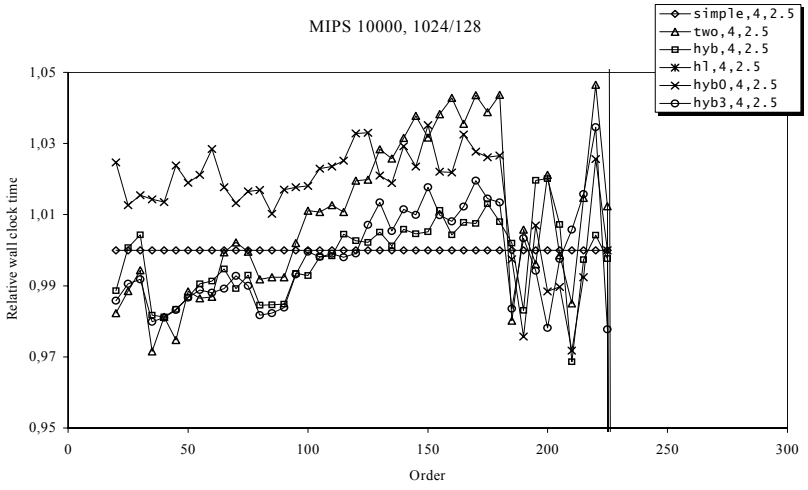


Chart C-36. Basic mergers, $(\alpha,d) = (4,2.5)$, MIPS 10000.

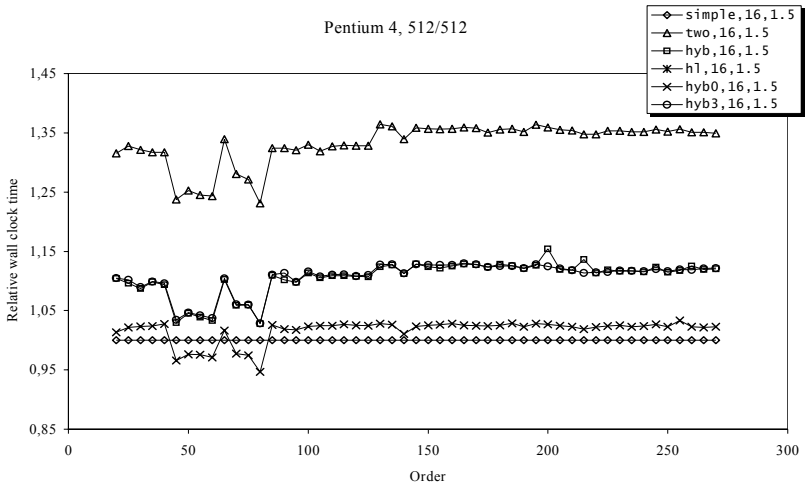


Chart C-37. Basic mergers, $(\alpha,d) = (16,1.5)$, Pentium 4.

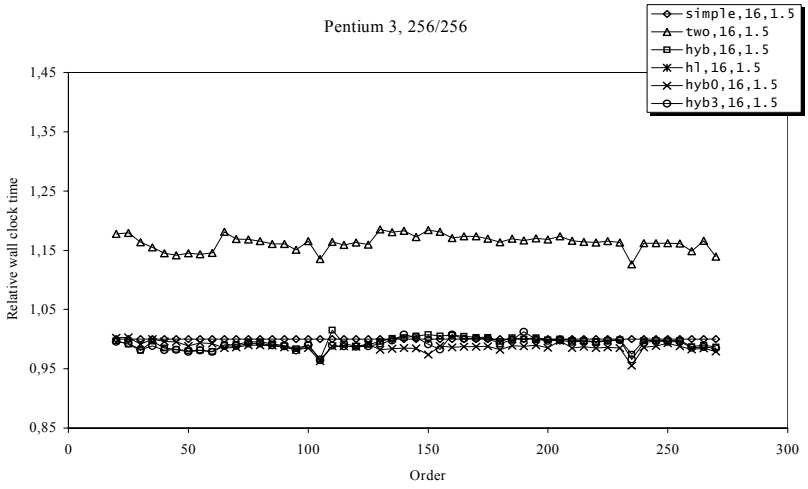


Chart C-38. Basic mergers, $(\alpha,d) = (16,1.5)$, Pentium 3.

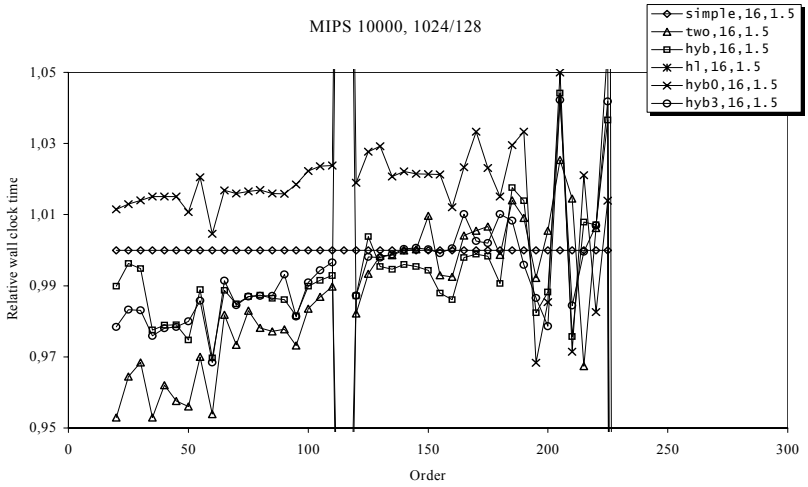


Chart C-39. Basic mergers, $(\alpha,d) = (16,1.5)$, MIPS 10000.

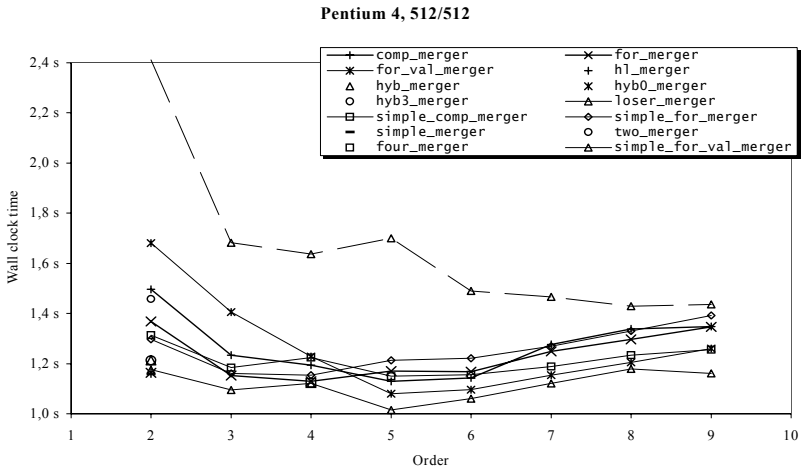


Chart C-40. Basic mergers, Pentium 4.

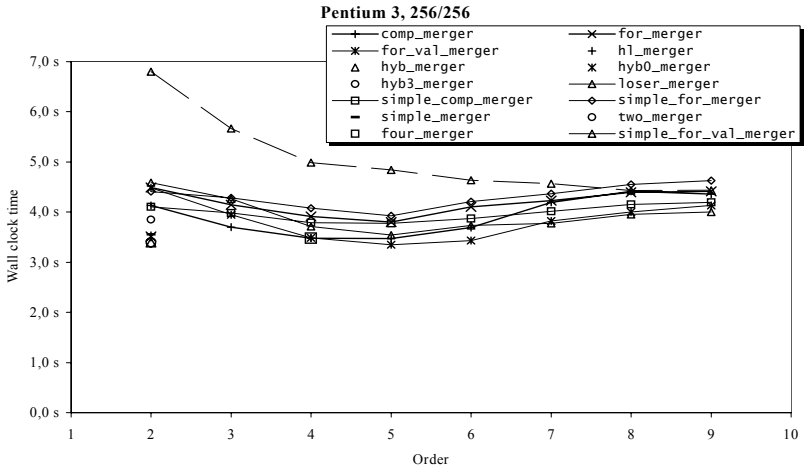


Chart C-41. Basic mergers, Pentium 3.

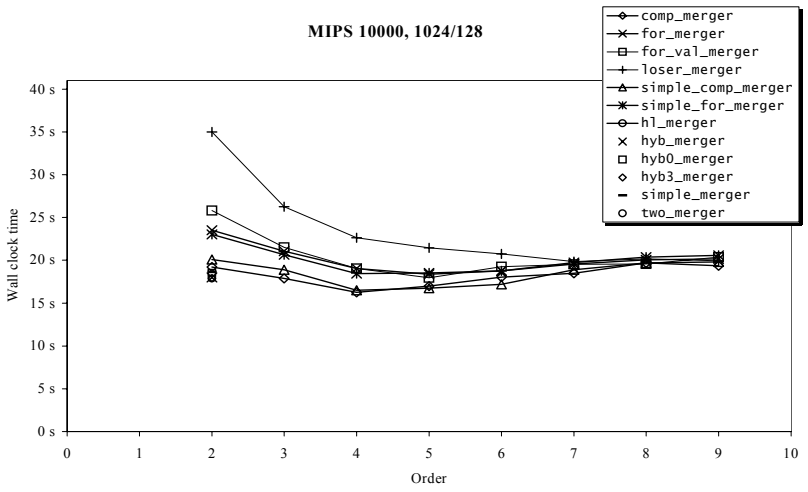


Chart C-42. Basic mergers, MIPS 10000.

C.3 Merger Caching

The results of the benchmarks described in Sections 5.4.1 and 5.4.2.

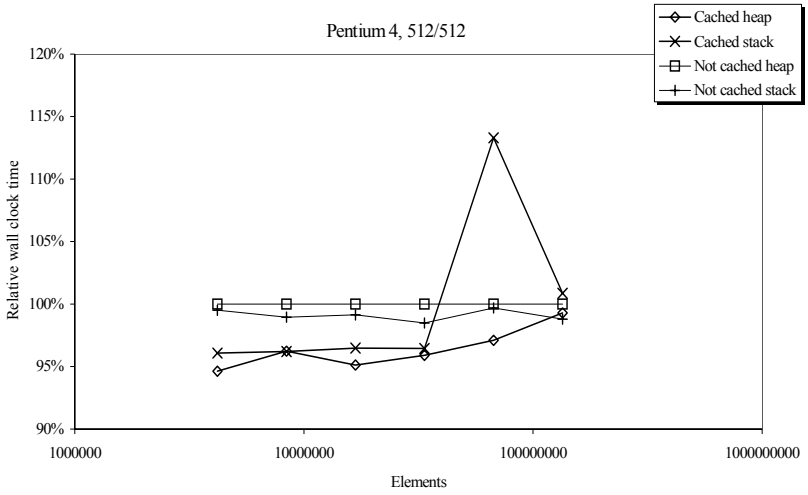


Chart C-43. Effects of merger caching, Pentium 4.

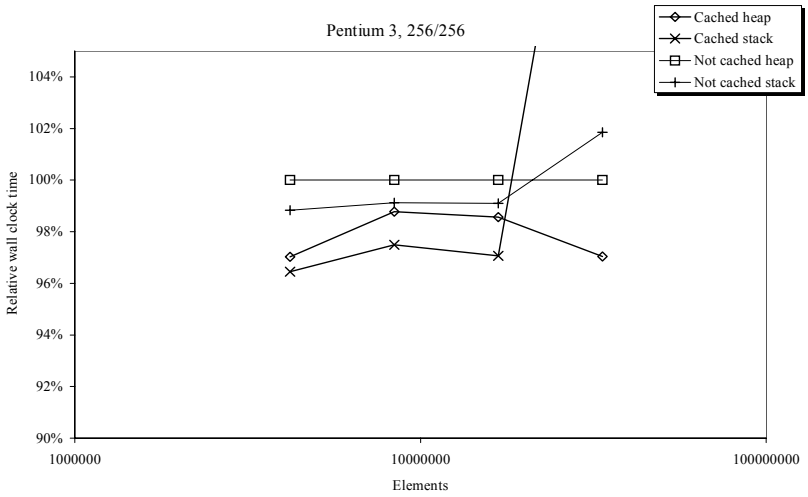


Chart C-44. Effects of merger caching, Pentium 3.

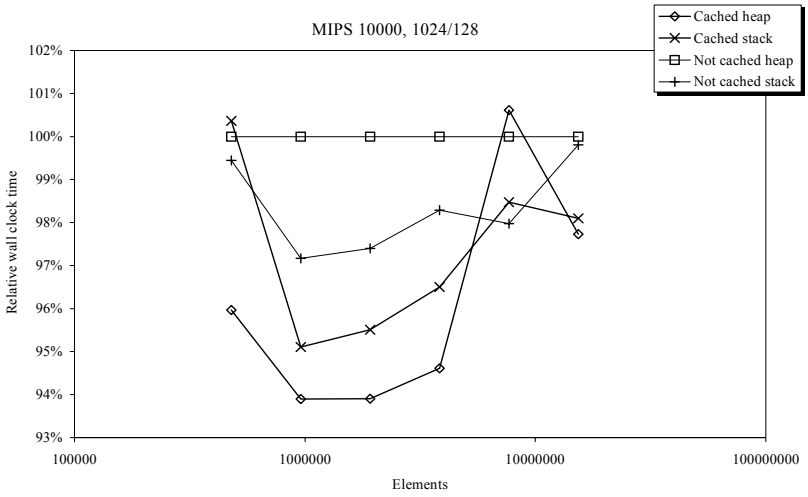


Chart C-45. Effects of merger caching, MIPS 10000.

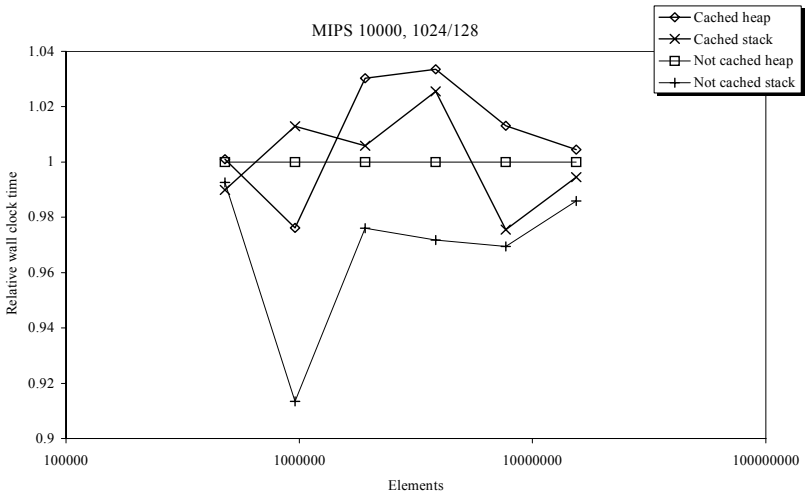


Chart C-46. Effects of merger caching, MIPS 10000, L2 cache misses.

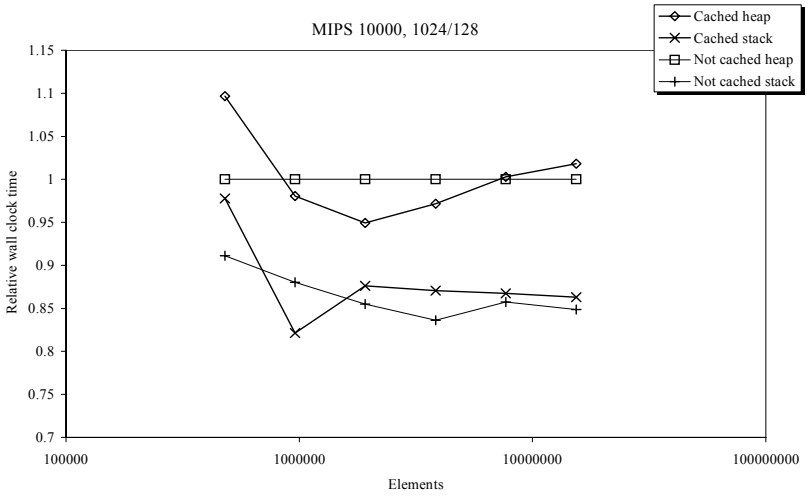


Chart C-47. Effects of merger caching, MIPS 10000, TLB misses.

C.4 Base Sorting Algorithms

The results of the benchmarks described in Section 5.4.3.

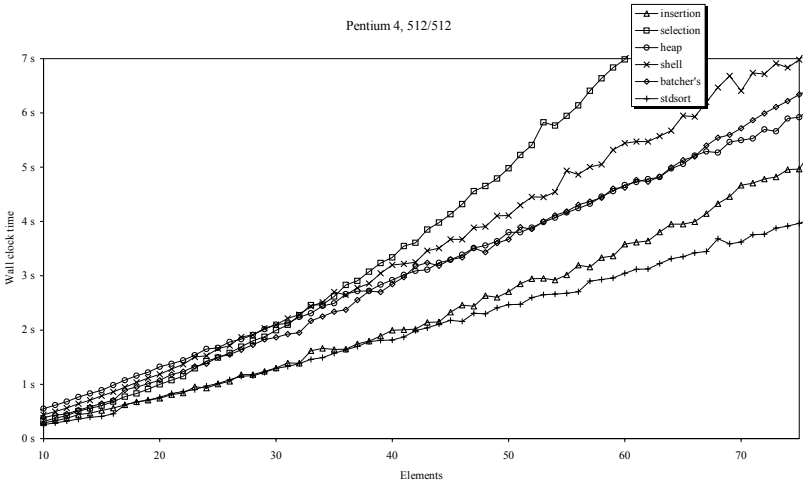


Chart C-48. Base sorting algorithms, Pentium 4.

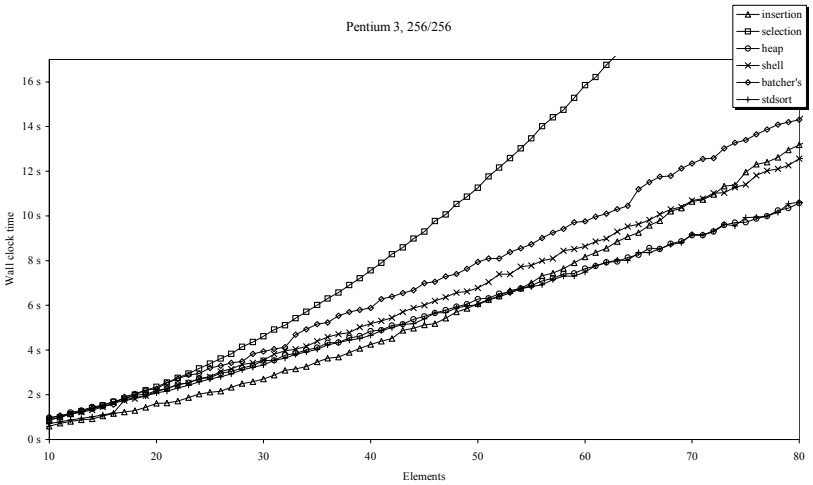


Chart C-49. Base sorting algorithms, Pentium 3.

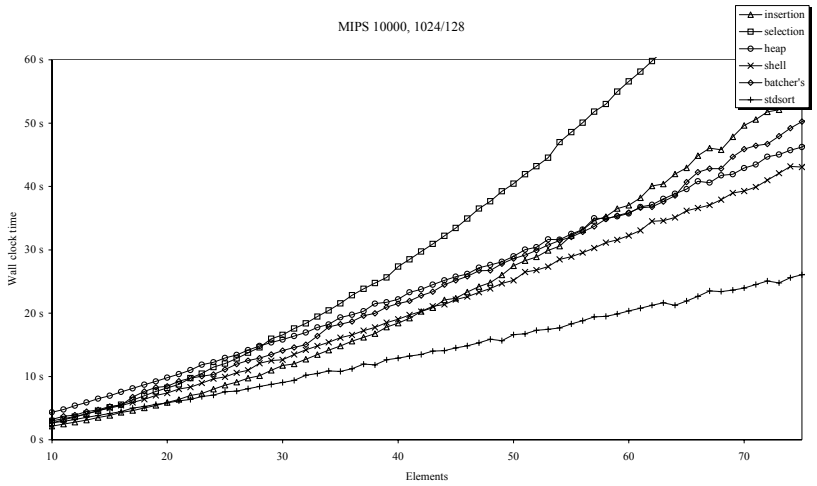


Chart C-50. Base sorting algorithms, MIPS 10000.

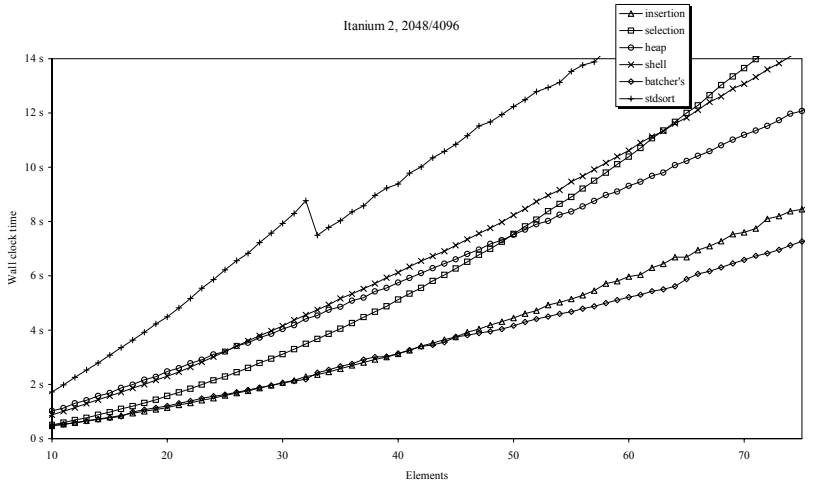


Chart C-51. Base sorting algorithms, Itanium 2.

C.5 Buffer Sizes

The results of the benchmarks described in Section 5.4.4.

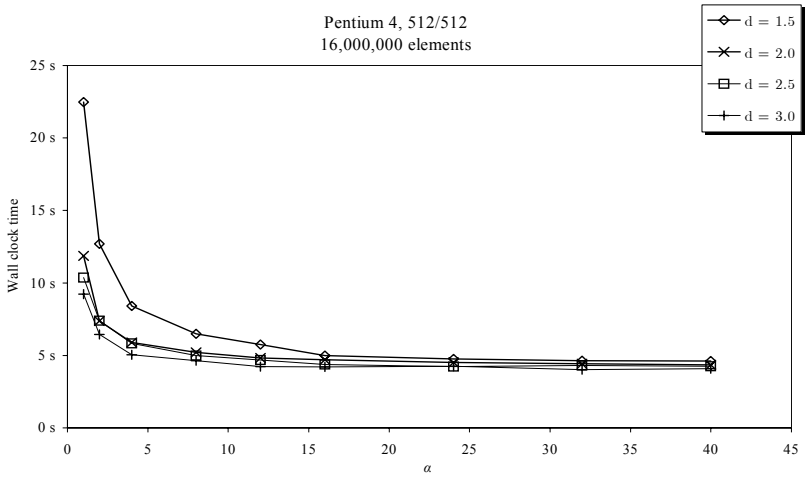


Chart C-52. Buffer parameters, sorting 16,000,000 elements on Pentium 4.

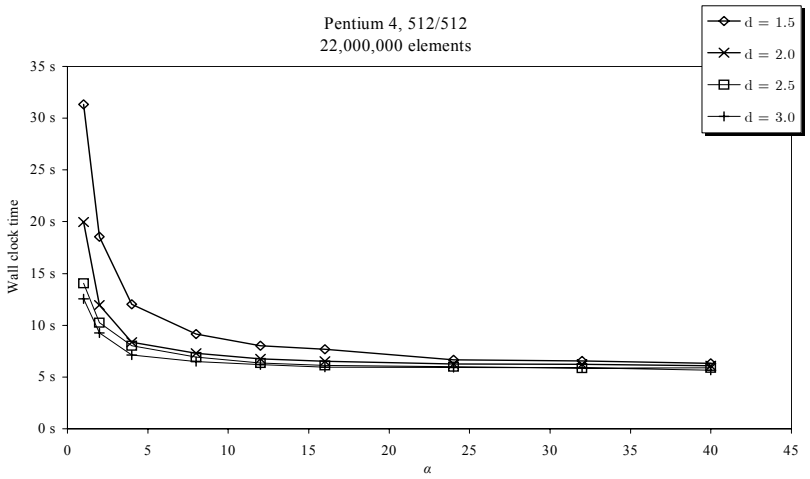


Chart C-53. Buffer parameters, sorting 22,000,000 elements on Pentium 4.

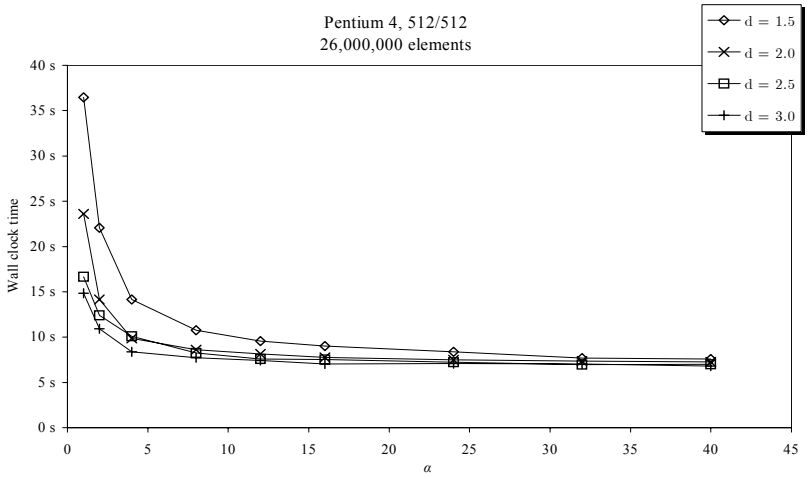


Chart C-54. Buffer parameters, sorting 26,000,000 elements on Pentium 4.

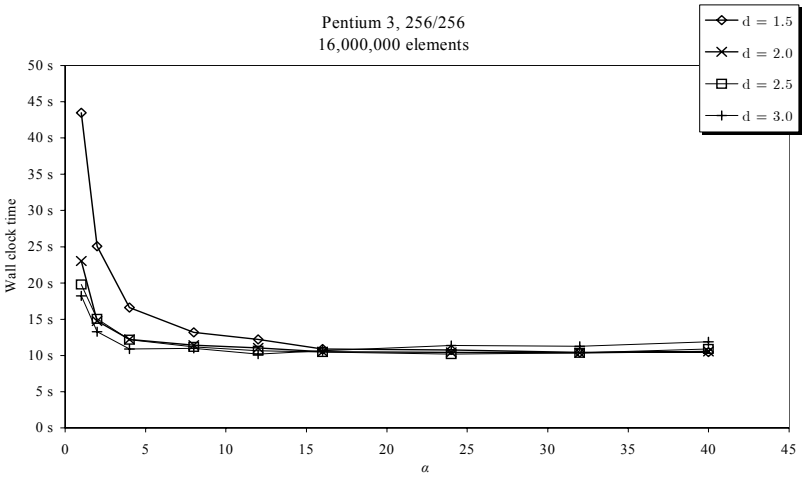


Chart C-55. Buffer parameters, sorting 16,000,000 elements on Pentium 3.

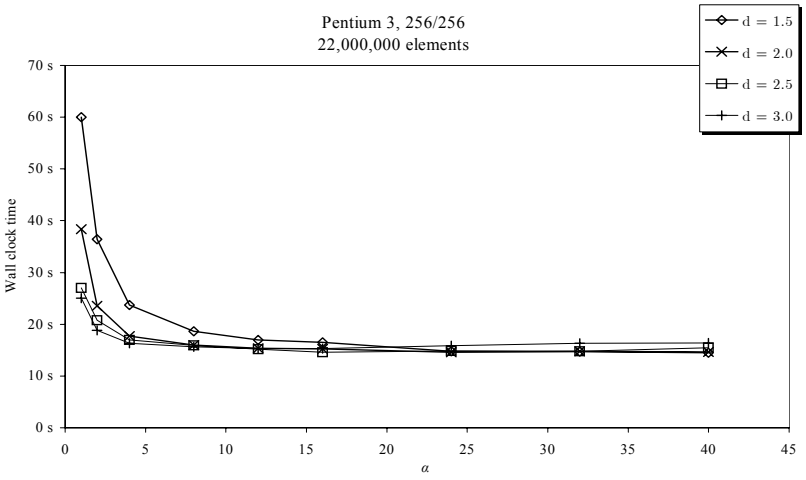


Chart C-56. Buffer parameters, sorting 22,000,000 elements on Pentium 3.

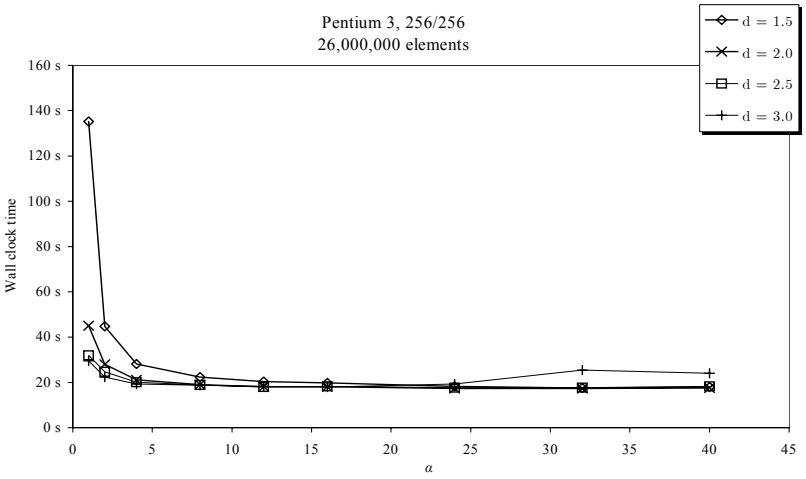


Chart C-57. Buffer parameters, sorting 26,000,000 elements on Pentium 3.

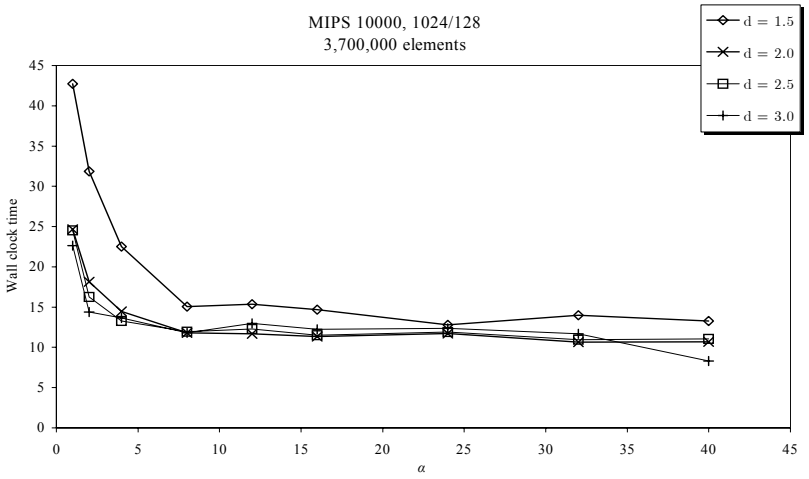


Chart C-58. Buffer parameters, sorting 3,700,000 elements on MIPS.

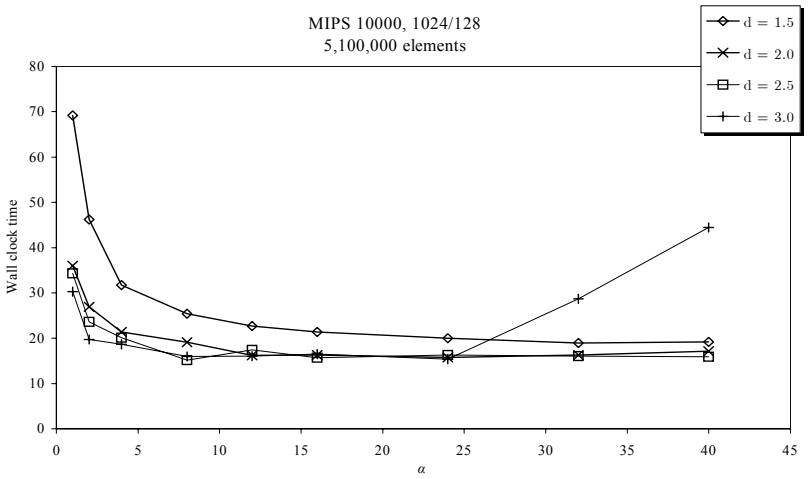


Chart C-59. Buffer parameters, sorting 5,100,000 elements on MIPS.

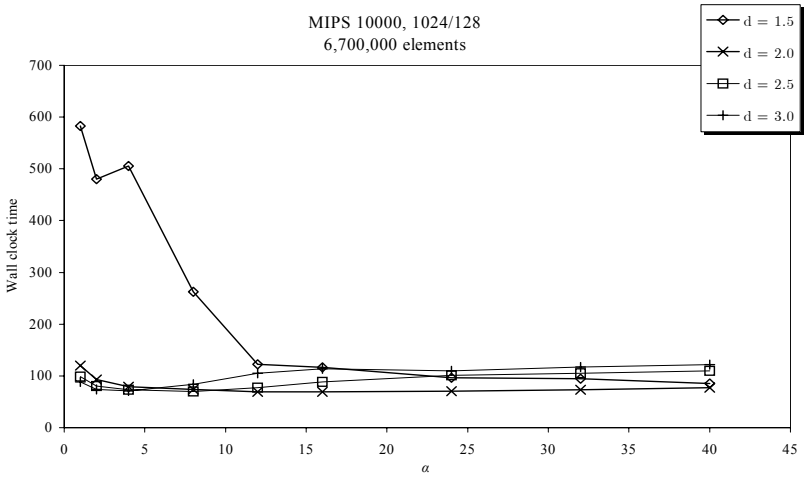


Chart C-60. Buffer parameters, sorting 6,700,000 elements on MIPS.

C.6 Straight Sorting

The results of the benchmarks described in Section 6.1 and 6.2.

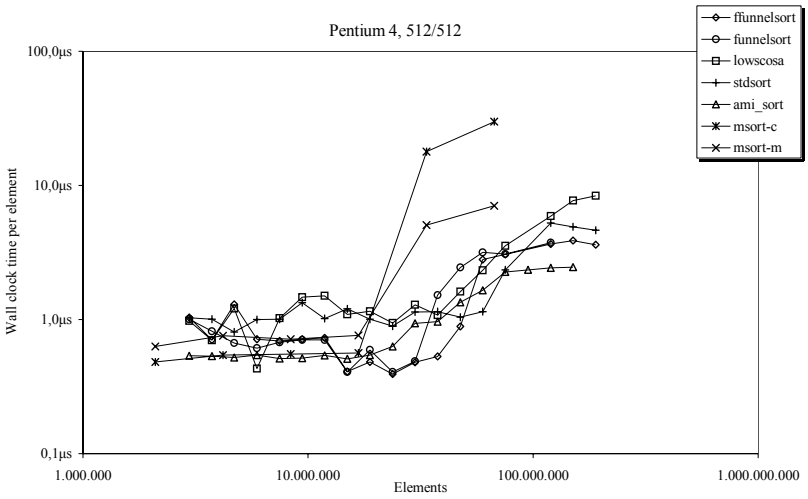


Chart C-61. Wall clock time sorting uniformly distributed pairs on Pentium 4.

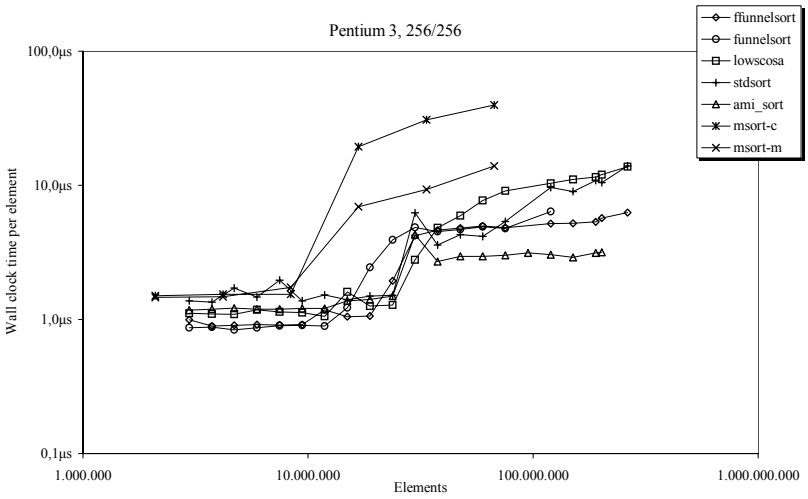


Chart C-62. Wall clock time sorting uniformly distributed pairs on Pentium 3.

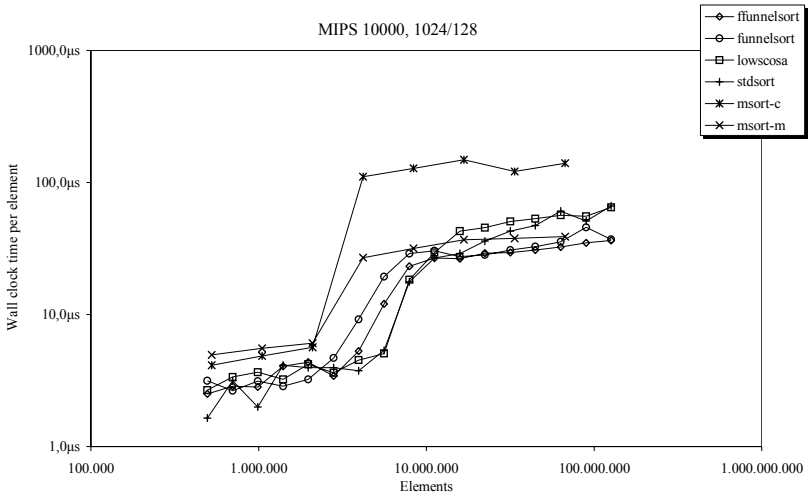


Chart C-63. Wall clock time sorting uniformly distributed pairs on MIPS 10000.

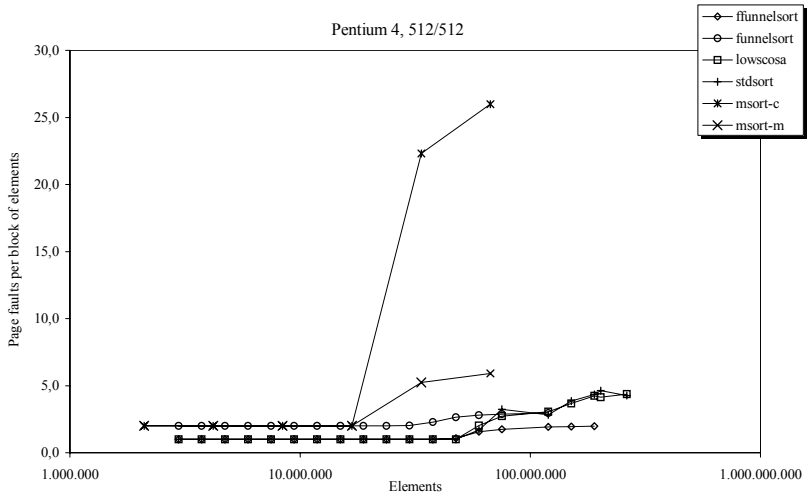


Chart C-64. Page faults sorting uniformly distributed pairs on Pentium 4.

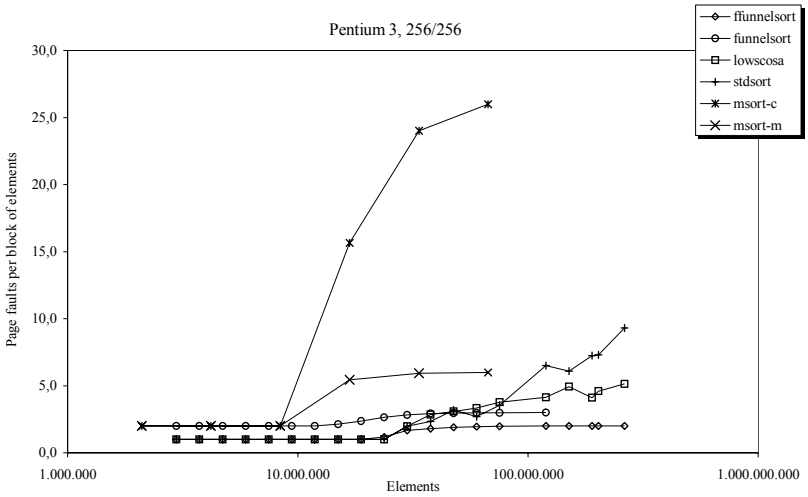


Chart C-65. Page faults sorting uniformly distributed pairs on Pentium 3.

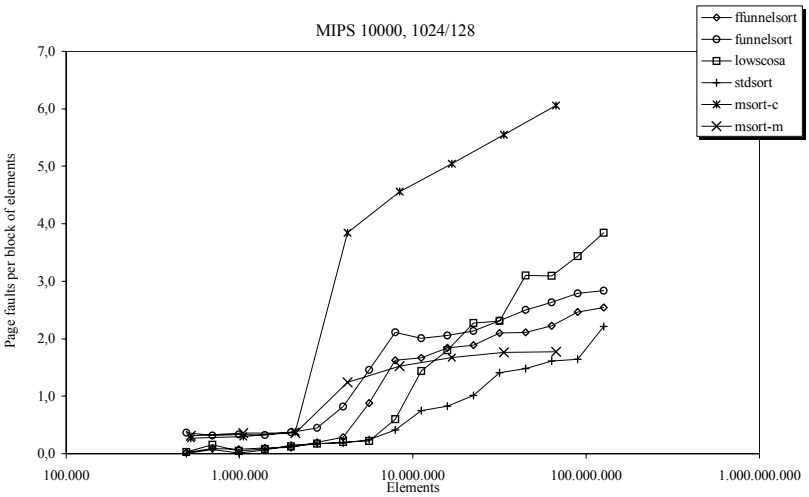


Chart C-66. Page faults sorting uniformly distributed pairs on MIPS 10000.

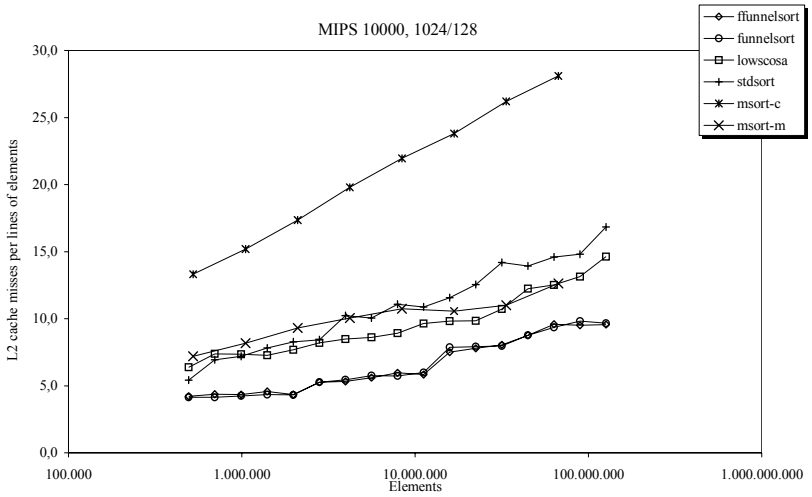


Chart C-67. Cache misses sorting uniformly distributed pairs on MIPS 10000.

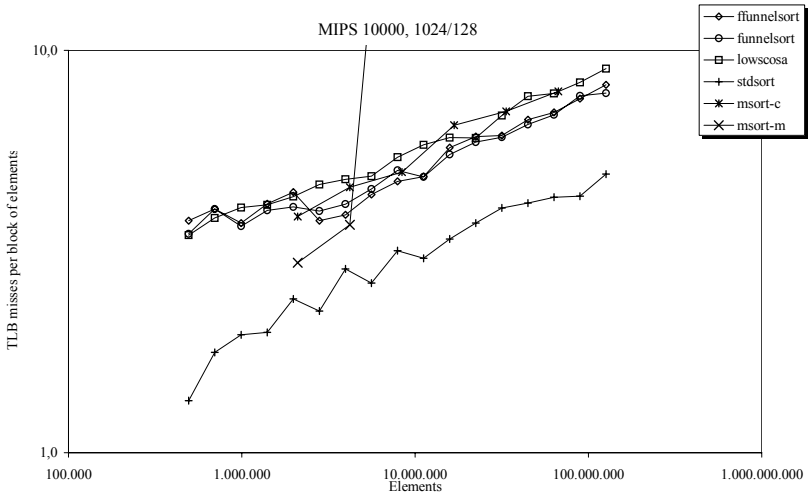


Chart C-68. TLB misses sorting uniformly distributed pairs on MIPS 10000.

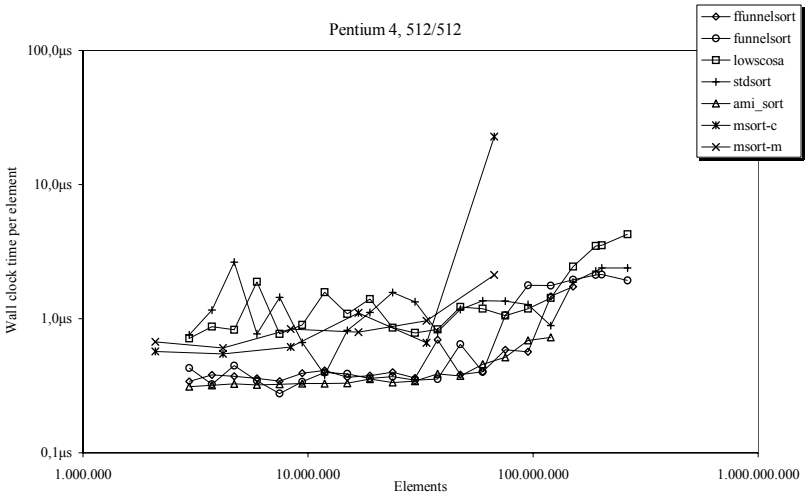


Chart C-69. Wall clock time sorting uniformly distributed integers on Pentium 4.

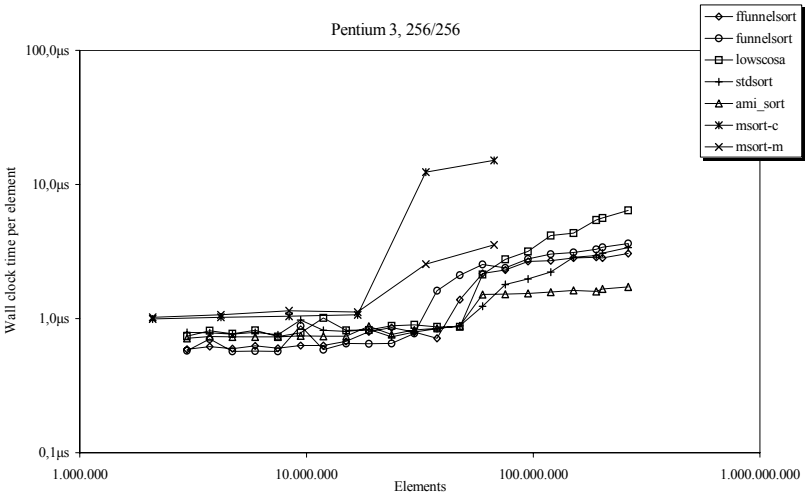


Chart C-70. Wall clock time sorting uniformly distributed integers on Pentium 3.

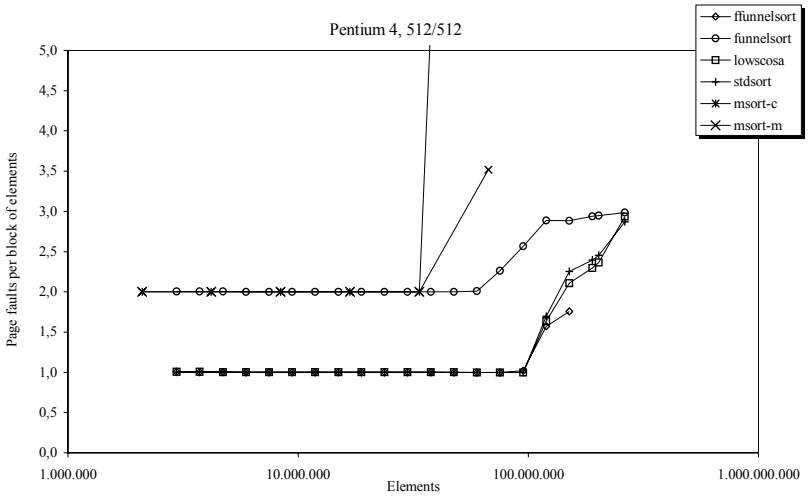


Chart C-71. Page faults sorting uniformly distributed integers on Pentium 4.

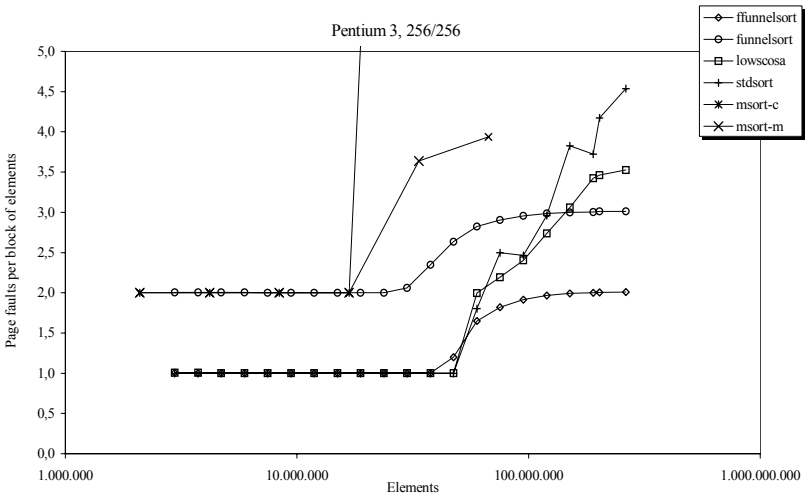


Chart C-72. Page faults sorting uniformly distributed integers on Pentium 3.

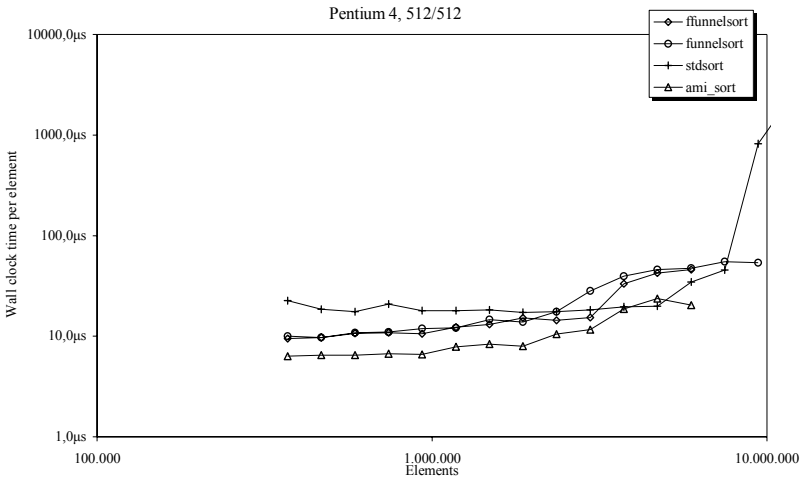


Chart C-73. Wall clock time sorting uniformly distributed records on Pentium 4.

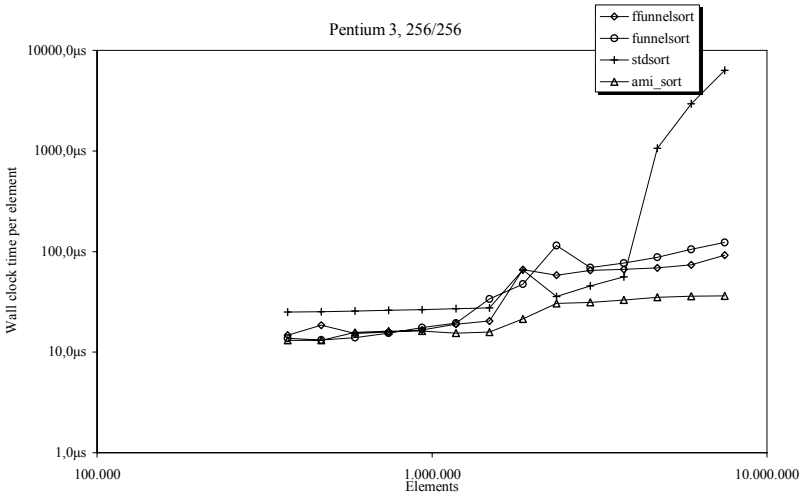


Chart C-74. Wall clock time sorting uniformly distributed records on Pentium 3.

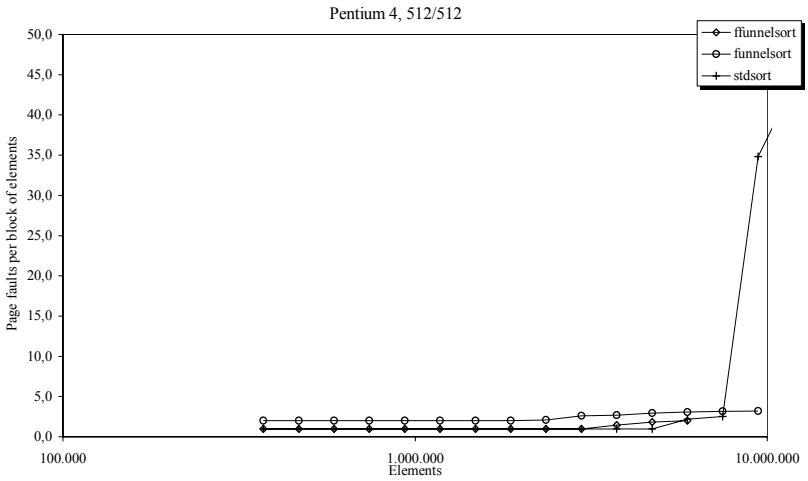


Chart C-75. Page faults sorting uniformly distributed records on Pentium 4.

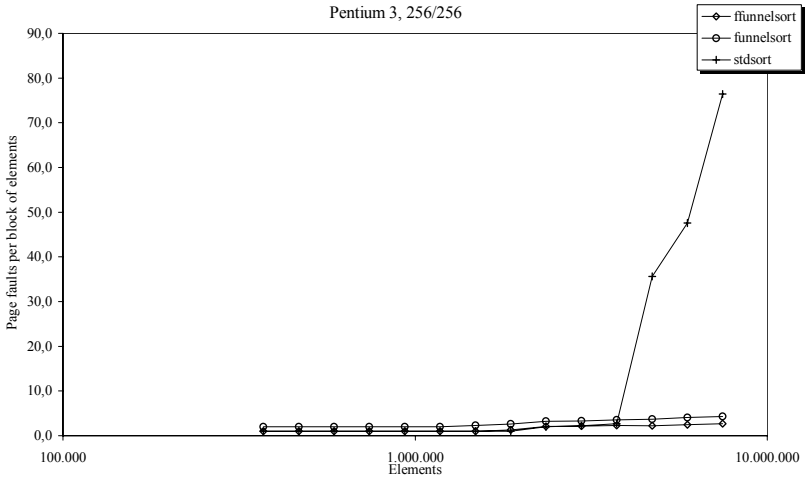


Chart C-76. Page faults sorting uniformly distributed records on Pentium 3.

C.7 Special Cases

The results of the benchmarks described in Sections 6.2 and 6.3.

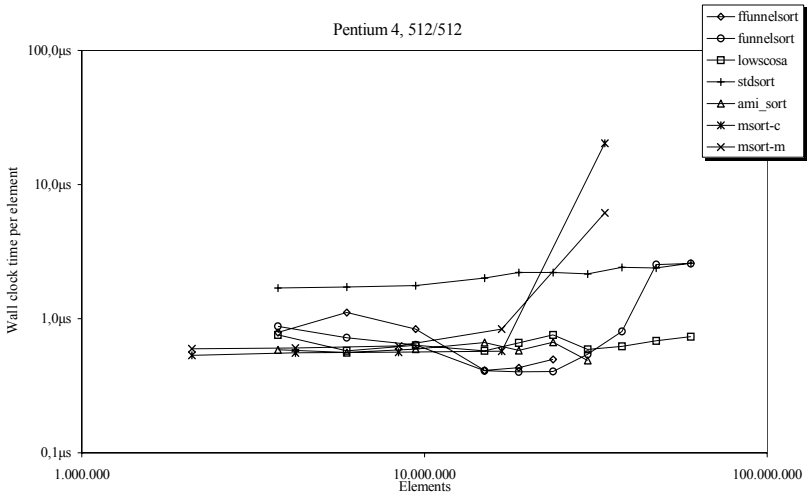


Chart C-77. Wall clock time sorting almost sorted pairs on Pentium 4.

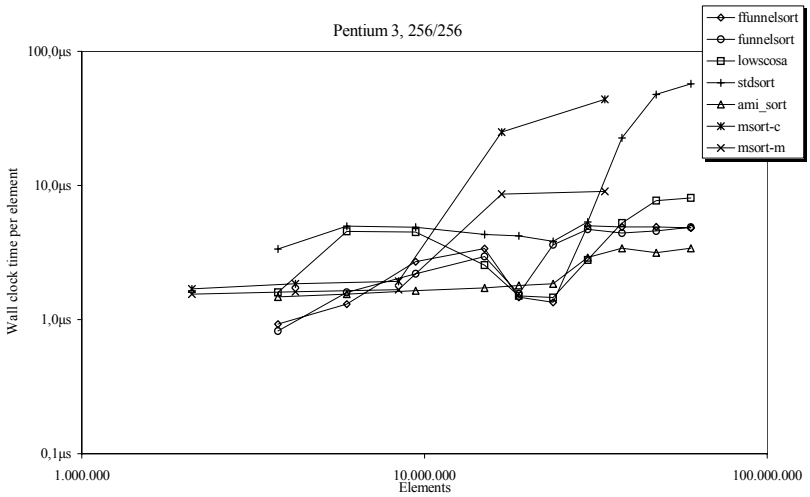


Chart C-78. Wall clock time sorting almost sorted pairs on Pentium 3.

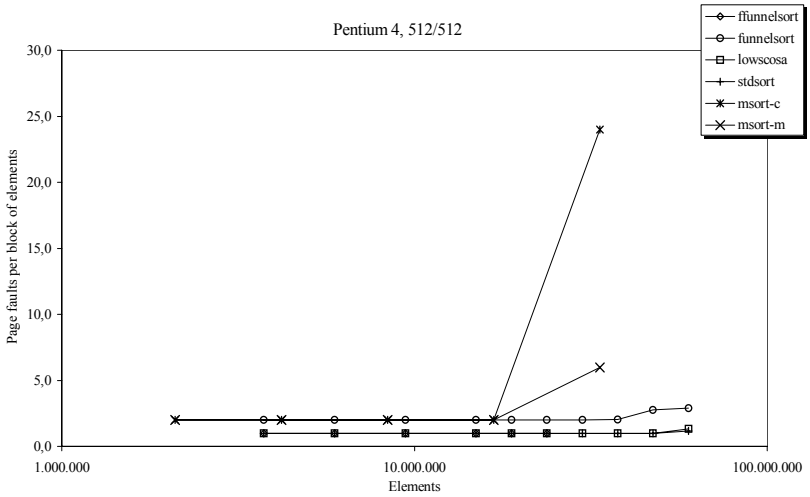


Chart C-79. Page faults sorting almost sorted pairs on Pentium 4.

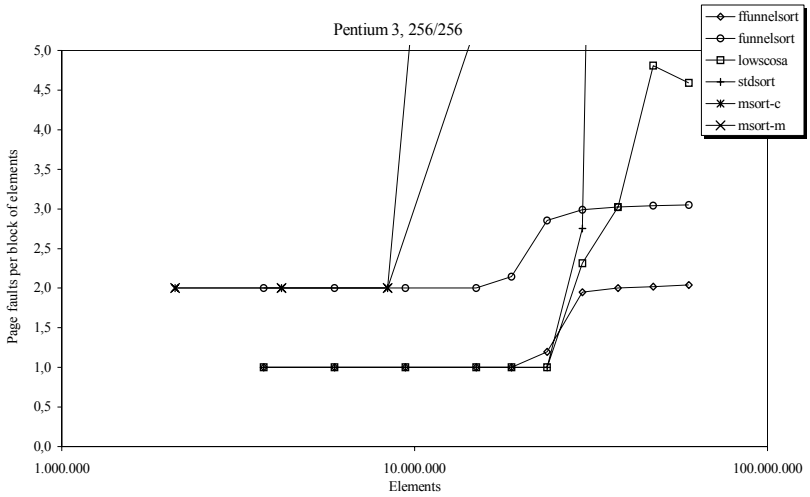


Chart C-80. Page faults sorting almost sorted pairs on Pentium 3.

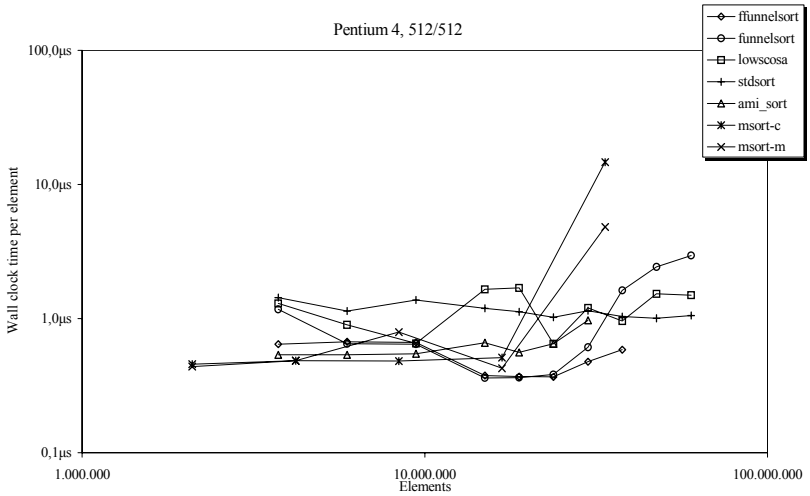


Chart C-81. Wall clock time sorting few distinct pairs on Pentium 4.

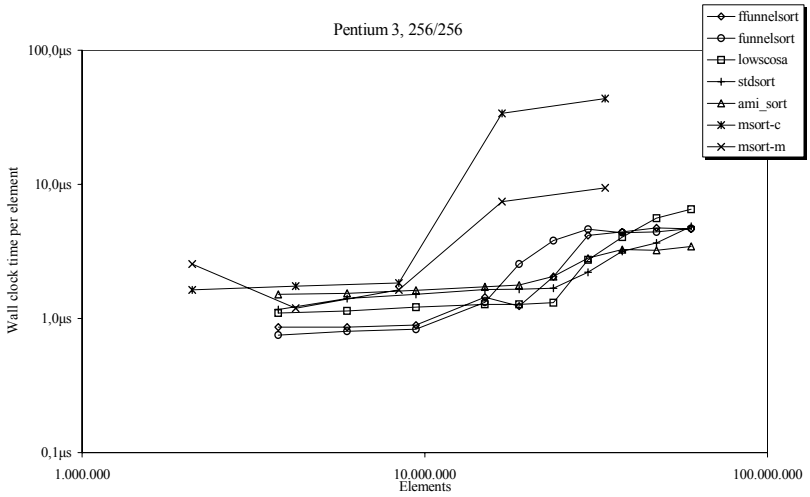


Chart C-82. Wall clock time sorting few distinct pairs on Pentium 3.

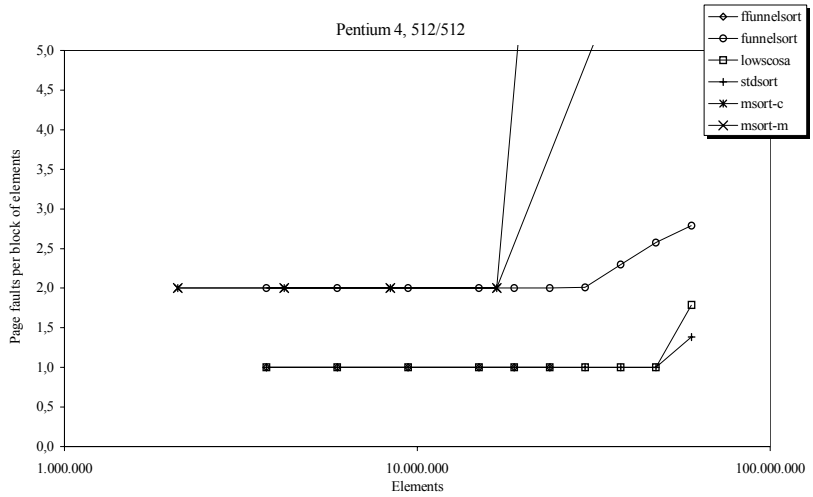


Chart C-83. Page faults sorting few distinct pairs on Pentium 4.

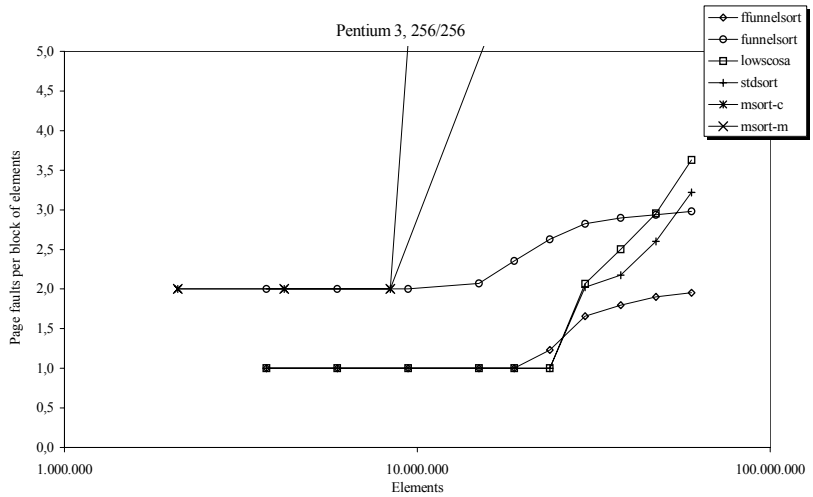


Chart C-84. Page faults sorting few distinct pairs on Pentium 3.

Bibliography

[ACV+00] RAJIV WICKREMESINGHE, LARS ARGE, JEFF CHASE, AND JEFFREY S. VITTER. Efficient Sorting Using Registers and Cache. ACM journal of Experimental Algorithmics, 2000.

[AMD02] AMD Athlon Processor x86 Code Optimization Guide. World Wide Web document, http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/22007.pdf, 2002.

[AV88] ALOK AGGARWAL AND JEFFERY S. VITTER. The input/output complexity of sorting and related problems. Communications of the ACM, 1988.

[BDFC00] MICHAEL A. BENDER, RICHARD COLE, ERIC DEMAINE, AND MARTIN FARACH-COLTON. Cache-oblivious b-trees. In Proceedings of the 10th Annual European Symposium on Algorithms, 2002.

[BF02a] GERTH S. BRODAL AND ROLF FAGERBERG. Cache oblivious distribution sweeping. In Proceedings of the 29th International Colloquium on Automata, Languages and Programming, 2002

[BF02b] GERTH S. BRODAL AND ROLF FAGERBERG. Funnel Heap – A cache oblivious priority queue. In Proceedings of the 13th Annual International Symposium on Algorithms and Computation, 2002.

[BFJ02] GERTH S. BRODAL, ROLF FAGERBERG, AND RICO JACOB. Cache Oblivious Search Trees via Binary Trees of Small Height, SODA, 2000.

[BF03] GERTH S. BRODAL AND ROLF FAGERBERG. On the limits of cache-obliviousness. Proceedings of 35th Annual ACM Symposium on Theory of Computation. To appear, 2003.

[BFP+73] MANUEL BLUM, ROBERT W. FLOYD, VAUGHAN PRATT, RONALD L. RIVEST, AND ROBERT E. TARJAN. Time bounds for selection. Journal of Computer and System Sciences, 1973.

[C++98] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. ISO/IEC 14882: Standard for the C++ Programming Language, 1998.

[CM99] ANDREAS CRAUSER AND KURT MEHLHORN. LEDA-SM A Platform for Secondary Memory Computation, Lecture Notes in Computer Science, 1999. See also http://www.mpi-sb.mpg.de/~crauser/leda-sm.html.

[DB03] Datamation Benchmark. Sort Benchmark Home Page, hosted by Microsoft. World Wide Web document, http://research.microsoft.com/barc/SortBenchmark/, 2003.

[Dem02] ERIK D. DEMAINE. Cache-oblivious Algorithms and Data Structures. Lecture notes, Massive Data Sets summer school, University of Aarhus, 2002.

[EKZ77] PETER VAN EMDE BOAS, ROB KAAS, AND JAN ZIJLSTRA. Design and Implementation of an Efficient Priority Queue. Mathematical Systems Theory 10, 1977.

[Flo72] ROBERT W. FLOYD. Permuting information in idealized two-level storage. Complexity of Computer Calculations, 1972.

[FLPR99] MATREO FRIGO, CHARLES E. LEISERSON, HARALD PROKOP, AND SHIDHAR RAMACHANDRAN. Cache-oblivious algorithms. Proceedings of the 40th Annual Symposium on Foundations of Computer Science, New York, 1999.

[HSU+01] GLENN HINTON, DAVE SAGER, MIKE UPTON, DARRELL BOGGS, DOUG CARMEAN, ALAN KYKER, AND PATRICE ROUSSEL. The Microarchitecture of the Pentium® 4 processor. World Wide Web document, http://www.intel.com/technology/itj/q12001/articles/art_2.htm, 2001.

[HK81] JIA.WEI HONG AND H. T. KUNG. I/O complexity: The red-blue pebble game. Proceedings of the 13th Annual ACM Symposium on Theory of Computation, Milwaukee, Wisconsin, 1981.

[HP96] JOHN L. HENNESSY AND DAVID A. PATTERSON. Computer Architecture: A Quatitative Approach, second edition. Morgan Kaufmann, 1996.

[Joh01] DAVID S. JOHNSON. A Theoretician's Guide to the Experimental Analysis of Algorithms. Proceedings of the 5th and 6th DIMACS Implementation Challenges. Goldwasser, Johnson, and McGeoch (eds), American Mathematical Society, 2001.

[Knu98] DONALD E. KNUTH. The Art of Computer Programming, Vol. 3: Sorting and Searching, second edition, Addison-Wesley, 1998.

[LFN02] RICHARD E. LADNER, RAY FORTNA, AND BAO-HOANG NGUYEN. A Comparison of Cache Aware and Cache Oblivious Static Search Trees Using Program Instrumentation, 2002.

[LL99] ANTHONY LAMARCA AND RICHARD E. LADNER. The Influence of Caches on the Performance of Sorting, Journal of Algorithms Vol. 31, 1999.

[MPS02] MIPS64™ Architecture for Programmers, Vol. 1: Introduction to the MIPS64™ Architecture. World Wide Web document, http://www.mips.com/content/Documentation/MIPSDocumentation/ProcessorArchitecture/doclibrary, 2002.

[MR01] CONRADO MARTÍNEZ AND SALVADOR ROURA. Optimal Sampling Strategies in Quicksort and Quickselect. SIAM Journal of Computing, 2001.

[Mus97] DAVID R. MUSSER. Introspective Sorting and Selection Algorithms. Software – Practice and Experience, vol. 9, nr. 8, 1997.

[NM95] STEFAN NÄHER AND KURT MEHLHORN. The LEDA Platform of Combinatorial and Geometric Computing, Communications of the ACM, 1995.

[Neu98] KARL-DIETRICH NEUBERT. The Flashsort1 algorithm. Dr. Dobb's journal, 123-125, 1998.

[OS02] JESPER H. OLSEN AND SØREN C. SKOV. Cache-Oblivious Algorithms in Practice, 2002.

[PAPI03] Performance application programming interface. World Wide Web document, http://icl.cs.utk.edu/projects/papi/, 2003.

[RR00] NAILA RAHMAN AND RAJEEV RAMAN. Analysing cache effects in distribution sorting, ACM journal of Experimental Algorithms, Vol. 5, 2000.

[Sav98] JOHN E. SAVAGE. Models of Computation. Addison Wesley Longman, Inc., 1998.

[Sed78] ROBERT SEDGEWICK. Implementing Quicksort Programs. Communications of the ACM, 21(10), 1978.

[ST85] DANIEL D. SLEATOR AND ROBERT E. TARJAN. Amortized efficiency of list update and paging rules. Communications of the ACM, 1985.

[Tan99] ANDREW S. TANENBAUM. Structured Computer Organization, 4th edition. Prentice-Hall Inc., 1999.

[Tan01] ANDREW S. TANENBAUM. Modern operating systems, 2nd edition. Prentice-Hall Inc., 2001.

[TPI02] Duke University. A Transparent Parallel I/O Environment, World Wide Web Document, http://www.cs.duke.edu/TPIE/, 2002.

[Vit01] JEFFREY S. VITTER. External Memory Algorithms and Data Structures: Dealing with Massive Data. ACM Computing Surveys, 2001.

[Wil64] JOHN W. J. WILLIAMS. Algorithm 232: Heapsort. Communications of the ACM, 1964.

[XZK00] LI XIAO, XIAODONG ZHANG, AND STEFAN A. KUBRICHT. Improving Memory Performance of Sorting Algorithms, ACM Journal of Experimental Algorithms, Vol. 5, 2000.

[Yea96] KENNETH C. YEAGER. The MIPS R10000 Superscalar Microprocessor. IEEE Micro, 16(2), 28-40, 1996.

Index

I

K

L

M

N

O

P

Q

R

S

T

V

W

X