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.

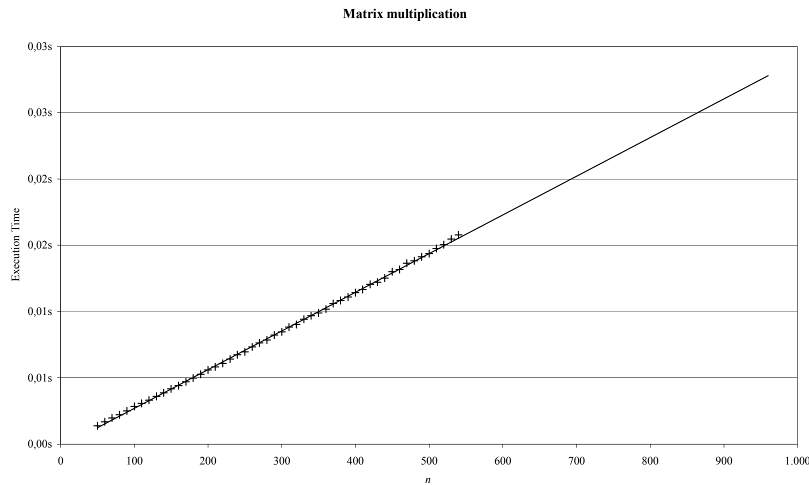

Figure 1-1: Average execution time of matrix multiplication.

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.

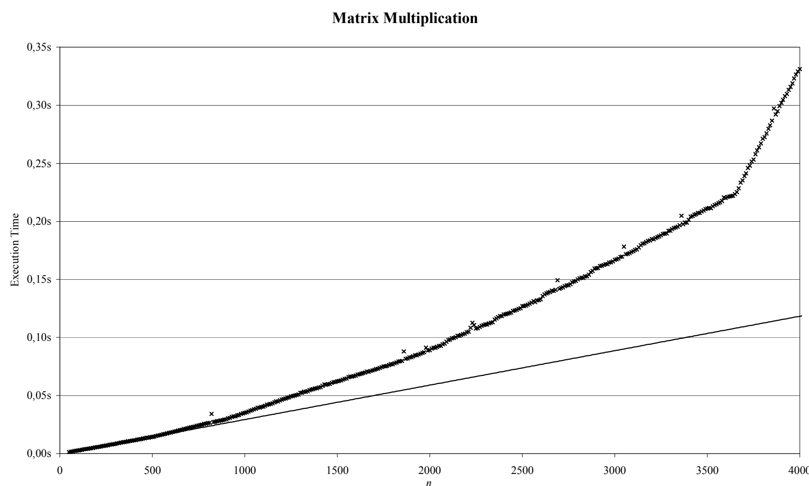

Figure 1-2: Average execution time of matrix multiplication for larger n.

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.

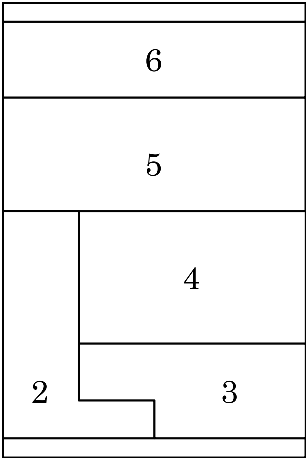

Figure 1-3. Structure of the thesis.

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.