Portable High-Performance Programs

by

Matteo Frigo

Laurea, Università di Padova (1992)
Dottorato di Ricerca, Università di Padova (1996)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1999

Author . . . .
Department of Electrical Engineering and Computer Science
June 23, 1999

Certified by . . . .
Charles E. Leiserson
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Portable High-Performance Programs

by

Matteo Frigo

Submitted to the Department of Electrical Engineering and Computer Science
on June 23, 1999, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

This dissertation discusses how to write computer programs that attain both high performance and portability, despite the fact that current computer systems have different degrees of parallelism, deep memory hierarchies, and diverse processor architectures.

To cope with parallelism portably in high-performance programs, we present the ***Cilk*** multithreaded programming system. In the Cilk-5 system, parallel programs scale up to run efficiently on multiple processors, but unlike existing parallel-programming environments, such as MPI and HPF, Cilk programs "scale down" to run on one processor as efficiently as a comparable C program. The typical cost of spawning a parallel thread in Cilk-5 is only between 2 and 6 times the cost of a C function call. This efficient implementation was guided by the ***work-first principle***, which dictates that scheduling overheads should be borne by the critical path of the computation and not by the work. We show how the work-first principle inspired Cilk's novel "two-clone" compilation strategy and its Dijkstra-like mutual-exclusion protocol for implementing the ready deque in the work-stealing scheduler.

To cope portably with the memory hierarchy, we present asymptotically optimal algorithms for rectangular matrix transpose, FFT, and sorting on computers with multiple levels of caching. Unlike previous optimal algorithms, these algorithms are ***cache oblivious***: no variables dependent on hardware parameters, such as cache size and cache-line length, need to be tuned to achieve optimality. Nevertheless, these algorithms use an optimal amount of work and move data optimally among multiple levels of cache. For a cache with size Z and cache-line length L where $Z = \Omega(L^2)$ the number of cache misses for an $m \times n$ matrix transpose is $\Theta(1 + mn/L)$. The number of cache misses for either an n-point FFT or the sorting of n numbers is $\Theta(1 + (n/L)(1 + \log_Z n))$. We also give a $\Theta(mnp)$-work algorithm to multiply an $m \times n$ matrix by an $n \times p$ matrix that incurs $\Theta(1 + (mn + np + mp)/L + mnp/L\sqrt{Z})$ cache faults.

To attain portability in the face of both parallelism and the memory hierarchy at the same time, we examine the ***location consistency*** memory model and the **BACKER** coherence algorithm for maintaining it. We prove good asymptotic bounds on the execution time of Cilk programs that use location-consistent shared memory.

To cope with the diversity of processor architectures, we develop the FFTW self-optimizing program, a portable C library that computes Fourier transforms. FFTW is unique in that it can automatically tune itself to the underlying hardware in order to achieve high performance. Through extensive benchmarking, FFTW has been shown to be typically faster than all other publicly available FFT software, including codes such as Sun's Performance Library and IBM's ESSL that are tuned to a specific machine. Most of the performance-critical code of FFTW was generated automatically by a special-purpose compiler written in Objective Caml, which uses symbolic evaluation and other compiler techniques to produce "codelets"—optimized sequences of C code that can be assembled into "plans" to compute a Fourier transform. At runtime, FFTW measures the execution time of many plans and uses dynamic programming to select the fastest. Finally, the plan drives a special interpreter that computes the actual transforms.

Thesis Supervisor: Charles E. Leiserson
Title: Professor of Computer Science and Engineering

Contents

Acknowledgements

This brief chapter is the most important of all. Computer programs will be outdated, and theorems will be shown to be imprecise, incorrect, or just irrelevant, but the love and dedition of all people who knowingly or unknowingly have contributed to this work is a lasting proof that life is supposed to be beautiful and indeed it is.

Thanks to Charles Leiserson, my most recent advisor, for being a great teacher. He is always around when you need him, and he always gets out of the way when you don't. (Almost always, that is. I wish he had not been around that day in Singapore when he convinced me to eat curried fish heads.)

I remember the first day I met Gianfranco Bilardi, my first advisor. He was having trouble with a computer, and he did not seem to understand how computers work. Later I learned that real computers are the only thing Gianfranco has trouble with. In any other branch of human knowledge he is perfectly comfortable.

Thanks to Arvind and Martin Rinard for serving on my thesis committee. Arvind and his student Jan-Willem Maessen acquainted me with functional programming, and they had a strong influence on my coding style and philosophy. Thanks to Toni Mian for first introducing me to Fourier transforms. Thanks to Alan Edelman for teaching me numerical analysis and algorithms. Thanks to Guy Steele and Gerry Sussman for writing the papers from which I learned what computer science is all about.

It was a pleasure to develop Cilk together with Keith Randall, one of the most talented hackers I have ever met. Thanks to Steven Johnson for sharing the burden of developing FFTW, and for many joyful moments. Volker Strumpen influenced many of my current thoughts about computer science as well as much of my personality. From him I learned a lot about computer systems. Members of the Cilk group were a constant source of inspiration, hacks, and fun. Over the years, I was honored to work with Bobby Blumofe, Guang-Ien Cheng, Don Dailey, Mingdong Feng, Chris Joerg, Bradley Kuszmaul, Phil Lisiecki, Alberto Medina, Rob Miller, Aske Plaat, Harald Prokop, Sridhar Ramachandran, Bin Song, Andrew Stark, and Yuli Zhou. Thanks to my officemates, Derek Chiou and James Hoe, for many hours of helpful and enjoyable conversations.

Thanks to Tom Toffoli for hosting me in his house when I first arrived to Boston. Thanks to

Irena Sebeda for letting me into Tom's house, because Tom was out of country that day. Thanks for Benoit Dubertret for being my business partner in sharing a house and a keg of beer, and for the good time we had during that partnership.

I wish to thanks all other people who made my stay in Boston enjoyable: Eric Chang, Nicole Lazo, Victor Luchangco, Betty Pun, Stefano Soatto, Stefano Totaro, Joel Villa, Carmen Young. Other people made my stay in Boston enjoyable even though they never came to Boston (proving that computers are good for something): Luca Andreucci, Alberto Cammozzo, Enrico Giordani, Gian Uberto Lauri, Roberto Totaro. Thanks to Andrea Pietracaprina and Geppino Pucci for helpful discussions and suggestions at the beginning of my graduate studies.

Thanks to Giuseppe (Pino) Torresin and the whole staff of Biomedin for their help and support during these five years, especially in difficult moments.

I am really grateful to Compaq for awarding me the Digital Equipment Corporation Fellowship. Additional financial support was provided by the Defense Advanced Research Projects Agency (DARPA) under Grants N00014-94-1-0985 and F30602-97-1-0270.

Many companies donated equipment that was used for the research described in this document. Thanks to SUN Microsystems Inc. for its donation of a cluster of 9 8-processor Ultra HPC 5000 SMPs, which served as the primary platform for the development of Cilk and of earlier versions of FFTW. Thanks to Compaq for donating a cluster of 7 4-processors AlphaServer 4100. Thanks to Intel Corporation for donating a four-processor Pentium Pro machine, and thanks to the Linux community for giving us a decent OS to run on it.

The Cilk and FFTW distributions use many tools from the GNU project, including `automake`, `texinfo`, and `libtool` developed by the Free Software Foundation. The `genfft` program was written using Objective Caml, a small and elegant language developed by Xavier Leroy. This dissertation was written on Linux using the TeX system by Donald E. Knuth, GNU Emacs, and various other free tools such as `gnuplot`, `perl`, and the `scm` Scheme interpreter by Aubrey Jaffer.

Finally, my parents Adriano and Germana, and my siblings Marta and Enrico deserve special thanks for their continous help and love. Now it's time to go home and stay with them again.

I would have graduated much earlier had not Sandra taken care of me so well. She was patient throughout this whole adventure.

Chapter 1

Portable high performance

This dissertation shows how to write computer programs whose performance is portable in the face of ***multiprocessors***, ***multilevel hierarchical memory***, and diverse ***processor architectures***.

1.1 The scope of this dissertation

Our investigation of portable high performance focuses on general-purpose shared memory multiprocessor machines with a memory hierarchy, which include uniprocessor PC's and workstations, symmetric multiprocessors (SMP's), and CC-NUMA machines such as the SGI Origin 2000. We are focusing on machines with shared memory because they are commonly available today and they are growing in popularity because they offer good performance, low cost, and a single system image that is easy to administer. Although we are focusing on shared-memory multiprocessor machines, some of our techniques for portable high performance could be applied to other classes of machines such as networks of workstations, vector computers, and DSP processors.

While superficially similar, shared-memory machines differ among each other in many ways. The most obvious difference is the degree of parallelism (i.e., the number of processors). Furthermore, platforms differ in the organization of the memory hierarchy and in their processor architecture. In this dissertation we shall learn theoretical and empirical approaches to write high-performance programs that are reasonably oblivious to variations in these parameters. These three areas by no means exhaust the full topic of portability in high-performance systems, however. For example, we are not addressing important topics such as portable performance in disk I/O, graphics, user interfaces, and networking. We leave these topics to future research.

1.1.1 Coping with parallelism

As multiprocessors become commonplace, we ought to write parallel programs that run efficiently both on single-processor and on multiprocessor platforms, so that a user can run a program to extract maximum efficiency from whatever hardware is available, and a software developer does not need to maintain both a serial and a parallel version of the same code. We ought to write these portable parallel programs, but we don't. Typically instead, a parallel program running on one processor is so much slower and/or more complicated than the corresponding serial program that people prefer to use two separate codes. The Cilk-5 multithreaded language, which I have designed and implemented together with Charles Leiserson and Keith Randall [58], addresses this problem. In Cilk, one can write parallel multithreaded programs that run efficiently on any number of processors, including 1, and are in most cases not significantly more complicated than the corresponding serial codes.

Cilk is a simple extension of the C language with fork/join parallelism. Portability of Cilk programs derives from the observation, based on "Brent's theorem" [32, 71], that any Cilk computation can be characterized by two quantities: its ***work*** T_1, which is the total time needed to execute the computation on one processor, and its ***critical-path length*** T_∞, which is the execution time of the computation on a computer with an infinite number of processors and a perfect scheduler (imagine God's computer). Work and critical-path are properties of the computation alone, and they do not depend on the number of processors executing the computation. In previous work, Blumofe and Leiserson [30, 25] designed Cilk's "work-stealing" scheduler and proved that it executes a Cilk program on P processors in time T_P, where

$$T_P \leq T_1/P + O(T_\infty) \; . \tag{1.1}$$

In this dissertation we improve on their work by observing that Equation (1.1) suggests both an efficient implementation strategy for Cilk and an algorithmic design that only focuses on work and critical path, as we shall now discuss.

In the current Cilk-5 implementation, a typical Cilk program running on a single processor is only less than 5% slower than the corresponding sequential C program. To achieve this efficiency, we aimed at optimizing the system for the common case, like much of the literature about compilers [124] and computer architectures [79]. Rather than understanding quantitatively the common case, mainly by studying the behavior of existing (and sometimes outdated) programs such as the SPEC benchmarks, the common-case behavior of Cilk is predicted by a theoretical analysis that culminates into the ***work-first principle***. Specifically, overheads in the Cilk system can be divided into work and critical-path overhead. The work-first principle states that Cilk incurs only work overhead in the common case, and therefore we should put effort in reducing it even at the expense of critical-path overhead. We shall derive the work-first principle from Equation (1.1) in Chapter 2, where we also show how this principle inspired a "two-clone" compilation strategy for Cilk and a Dijkstra-like [46] work-stealing protocol that does not use locks in the common case.

With an efficient implementation of Cilk and a performance model such as Equation (1.1), we can now design portable high-performance multithreaded algorithms. Typically in Cilk, these algorithms have a ***divide-and-conquer*** flavor. For example, the canonical Cilk matrix multiplication program is recursive. To multiply 2 matrices of size $n \times n$, it splits each input matrix into 4 parts of size $n/2 \times n/2$, and it computes 8 matrix products recursively. (See Section 2.4.) In Cilk, even loops are typically expressed as recursive procedures, because this strategy minimizes the critical path of the program. To see why, consider a loop that increments every element of an array A of length n. This program would be expressed in Cilk as a recursive procedure that increments $A[0]$ if $n = 1$, and otherwise calls itself recursively to increment the two halves of A in parallel. This procedure performs $\Theta(n)$ work, since the work of the recursion grows geometrically and is dominated by the n leaves, and the procedure has a $\Theta(\lg n)$ critical path, because with an infinite number of processors we reach the leaves of the recursion in time $\Theta(\lg n)$, and all leaves can be computed in parallel. The naive implementation that forks n threads in a loop, where each thread increments one array element, is not as good in the Cilk model, because the last thread cannot be created until all previous threads have been, yielding a critical path proportional to n.

Besides being high-performance, Cilk programs are also portable, because they do not depend on the value of P. Cilk shares this property with functional languages such as Multilisp [75], Mul-T [94], Id [119], and data-parallel languages such as NESL [23], ZPL [34], and High Performance Fortran [93, 80]. Among these languages, only NESL and ZPL feature an algorithmic performance model like Cilk, and like Cilk, ZPL is efficient in practice [116]. The data-parallel style encouraged by NESL and ZPL, however, can suffer large performance penalties because it introduces temporary arrays, which increase memory usage and pollute the cache. Compilers can eliminate these temporaries with well-understood analyses [100], but the analysis is complicated and real compilers are not always up to this task [116]. The divide-and-conquer approach of Cilk is immune from these difficulties, and allows a more natural expression of irregular problems. We will see another example of the importance of divide and conquer for portable high performance in Section 1.1.2 below.

1.1.2 Coping with the memory hierarchy

Modern computer systems are equipped with a ***cache***, or fast memory. Computers typically have one or more levels of cache, which constitute the ***memory hierarchy***, and any programming system must deal with caches if it hopes to achieve high performance. To understand how to program caches efficiently and portably, in this dissertation we explore the idea of ***cache obliviousness***. Although a cache-oblivious algorithm does not "know" how big the cache is and how the cache is partitioned into "cache lines," these algorithms nevertheless use the cache asymptotically as efficiently as their cache-aware counterparts. In Chapter 3 we shall see cache-oblivious algorithms for matrix transpose and multiplication, FFT, and sorting. For problems such as sorting where lower bounds on execution time and "cache complexity" are known, these cache-oblivious algorithms are optimal in both respects.

A key idea for cache-oblivious algorithms is again ***divide and conquer***. To illustrate cache obliviousness, consider again a divide and conquer matrix multiplication program that multiplies two square matrices of size $n \times n$. Assume that initially n is big, so that the problem cannot be solved fully within the cache, and therefore some traffic between the cache and the slow main memory is necessary. The program partitions a problem of size n into 8 subproblems of size $n/2$ recursively, until $n = 1$, in which case it computes the product directly. Even though the initial array is too big to fit into cache, at some point during the recursion n reaches some value n_0 so small that two matrices of size $n_0 \times n_0$ can be multiplied fully within the cache. The program is not aware of this transition and it continues the recursion down to $n = 1$, but the cache system is built in such a way that it loads every element of the $n_0 \times n_0$ subarrays only once from main memory. With the appropriate assumptions about the behavior of the cache, this algorithm can be proven to use the cache asymptotically optimally, even though it does not depend on parameters such as the size of the cache. (See Chapter 3.) An algorithm does not necessarily use the cache optimally just because it is divide-and-conquer, of course, but in many cases the recursion can be designed so that the algorithm is (asymptotically) optimal no matter how large the cache is.

How can I possibly advocate recursion instead of loops for high performance programs, given that procedure calls are so expensive? I have two answers to this objection. First, procedure calls are not *too* expensive, and the overhead of the recursion is amortized as soon as the leaves of the recursion perform enough work. I have coded the procedure that adds 1 to every element of an array using both a loop and a full recursion. The recursive program is about 8 times slower than the loop on a 143-MHz UltraSPARC. If we unroll the leaves of the recursion so that each leaf performs about 100 additions, the difference becomes less than 10%. To put things in perspective, 100 additions is roughly the work required to multiply two 4×4 matrices or to perform a 16-point Fourier transform. Second, we should keep in mind that current processors and compilers are optimized for loop execution and not for recursion, and consequently procedure calls are relatively more expensive than they could be if we designed systems explicitly to support efficient recursion. Since divide and conquer is so advantageous for portable high-performance programs, we should see this as a research opportunity to investigate architectural innovations and compiler techniques that reduce the cost of procedure calls. For example, we need compilers that unroll recursion in the same way current compilers unroll loops.

Cache-oblivious algorithms are designed for an ***ideal cache***, which is fully associative (objects can reside anywhere in the cache) and features an optimal, omniscient replacement policy. In the same way as a Cilk parallel algorithm is characterized by its work and critical-path length, a cache-oblivious algorithm can be characterized by its work W and by its ***cache complexity*** $Q(Z, L)$, which measures the traffic between the cache and the main memory when the cache contains Z words and it is partitioned into "lines" of length L. This theoretical framework allows algorithmic design for the range (Z, L) of interest.

Our understanding of cache obliviousness is somewhat theoretical at this point, since today's computers do not feature ideal caches. Nevertheless, the ideal-cache assumptions are satisfied in many cases. Consider for example the compilation of straight-line code with many (local) variables, more than can fit into the register set of a processor. We can view the registers as the "cache" and the rest of the memory as "main memory." The compiler faces the problem of allocating variables to registers so as to minimize the transfers between registers and memory, that is, the number of "register spills" [115]. Because the whole sequences of accesses is known in advance, the compiler can implement the optimal replacement strategy from [18], which replaces the register accessed farthest in the future. Consequently, with a cache-oblivious algorithm and a good compiler, one can write a single piece of C code that minimizes the traffic between registers and memory in such a way that the same code is (asymptotically) optimal for any number of CPU registers. I have used this idea in the FFTW "codelet generator" (see Chapter 6), which generates cache-oblivious fast Fourier transform programs.

1.1.3 Coping with parallelism and memory hierarchy together

What happens when we parallelize a cache-oblivious algorithm with Cilk? The execution-time upper bound from [25] (that is, Equation (1.1)) does not hold in the presence of caches, because the proof does not account for the time spent in servicing cache misses. Furthermore, cache-oblivious algorithms are not necessarily cache-optimal when they are executed in parallel, because of the communication among caches.

In this dissertation, we combine the theories of Cilk and of cache obliviousness to provide a performance bound similar to Equation (1.1) for Cilk programs that use hierarchical shared memory. To prove this bound, we need to be precise about how we want memory to behave (the "memory model"), and we must specify a protocol that maintains such a model. This dissertation presents a memory model called ***location consistency*** and the BACKER coherence algorithm for maintaining it. If BACKER is used in conjunction with the Cilk scheduler, we derive a bound on the execution time similar to Equation (1.1), but which takes the cache complexity into account. Specifically, we prove that a Cilk program with work T_1, critical path T_∞, and cache complexity $Q(Z, L)$ runs on P processors in expected time

$$T_P = O((T_1 + \mu Q(Z, L))/P + \mu Z T_\infty / L) ,$$

where μ is the cost of transferring one cache line between main memory and the cache. As in Equation (1.1), the first term $T_1 + \mu Q(Z, L)$ is the execution time on one processor when cache effects are taken into account. The second term $\mu Z T_\infty / L$ accounts for the overheads of parallelism. Informally, this term says that we might have to refill the cache from scratch from time to time, where each refill costs time $\mu Z/L$, but this operation can happen at most T_∞ times on average. Although this model is simplistic, and it does not account for the fact that the service time is not constant in practice (for example, on CC-NUMA machines), Cilk with BACKER is to my knowledge the only system that provides performance bounds accounting for work, critical path, and cache complexity.

Location consistency is defined within a novel ***computation-centric*** framework on memory models. The implications of this framework are not directly relevant to the main point of this dissertation, which is how to write portable fast programs, but I think that the computation-centric framework is important from a "cultural" perspective, and therefore in Chapter 5 I have included a condensed version of the computation-centric theory I have developed elsewhere [54].

1.1.4 Coping with the processor architecture

We personally like Brent's algorithm for univariate minimization, as found on pages 79–80 of his book "Algorithms for Minimization Without Derivatives." It is pretty reliable and pretty fast, but we cannot explain how it works.

(Gerald Jay Sussman)

While work, critical path, and cache complexity constitute a clean high-level algorithmic characterization of programs, and while the Cilk theory is reasonably accurate in predicting the performance of parallel programs, a multitude of real-life details are not captured by the simple theoretical analysis of Cilk and of cache-oblivious algorithms. Currently we lack good models to analyze the dependence of algorithms on the virtual memory system, the associativity of caches, the depth of a processor pipeline, the number and the relative speeds of functional units within a processor, out-of-order execution, branch predictors, not to mention busses, interlocks, prefetching instructions, cache coherence, delayed branches, hazard detectors, traps and exceptions, and the aggressive code transformations that compilers operate on programs. We shall refer to these parameters generically as "processor architecture." Even though compilers are essential to any high-performance system, imagine for now that the compiler is part of some black box called "processor" that accepts our program and produces the results we care about.

The behavior of "processors" these days can be quite amazing. If you experiment with your favorite computer, you will discover that performance is not additive—that is, the execution time of a program is not the sum of the execution time of its components—and it is not even monotonic. For example, documented cases exist [95] where adding a "no-op" instruction to a program doubles its speed, a phenomenon caused by the interaction of a short loop with a particular implementation of branch prediction. As another example, the Pentium family of processors is much faster at loading double precision floating-point numbers from memory if the address is a multiple of 8 (I have observed a factor of 3 performance difference sometimes). Nevertheless, compilers like `gcc` do not enforce this alignment because it would break binary compatibility with existing 80386 code, where the alignment was not important for performance. Consequently, your program might become suddenly fast or slow when you add a local variable to a procedure. While it is unfortunate that the system as a whole exhibits these behaviors, we cannot blame processors: The architectural features that cause these anomalies are the very source of much of the processor performance. In current processor architectures we gave away understandable designs to buy performance—a pact with the devil [107] perhaps, but a good deal nonetheless.

Since we have no good model of processors, we cannot design "pipeline-oblivious" or "compiler-oblivious" algorithms like we did for caches. Nevertheless, we can still write portable high-performance programs if we adopt a "closed loop" approach. Our previous techniques were open-loop, and programs were by design oblivious to the number of processors and the cache. To cope with processors architectures, we will write closed-loop programs capable of determining their own performance and of adjusting their behavior to the complexity of the environment.

To explore this idea, I have developed a ***self-optimizing program*** that can measure its own execution speed to adapt itself to the "processor." ***FFTW*** is a comprehensive library of fast C routines for computing the ***discrete Fourier transform*** (DFT) in one or more dimensions, of both real and complex data, and of arbitrary input size. FFTW automatically adapts itself to the machine it is running on so as to maximize performance, and it typically yields significantly better performance than all other publicly available DFT software. More interestingly, while retaining complete portability, FFTW is competitive with or faster than proprietary codes, such as Sun's Performance Library and IBM's ESSL library, which are highly tuned for a single machine.

In order to adapt itself to the hardware, FFTW uses the property that the computation of a Fourier transform can be decomposed into subproblems, and this decomposition can typically be accomplished in many ways. FFTW tries many different decompositions, it *measures* their execution time, and it remembers the one that happens to run faster on a particular machine. FFTW does not attempt to build a performance model and to predict the performance of a given decomposition, because all my attempts to build a precise enough performance model to this end have failed. Instead, by measuring its own execution time, FFTW approaches portability in a closed loop, end-to-end fashion, and it compensates for our lack of understanding and for the imprecision of our theories.

FFTW's portability is enabled by the extensive use of ***metaprogramming***. About 95% of the FFTW system is comprised of ***codelets***, which are optimized sequences of C code that compute subproblems of a Fourier transform. These codelets were generated automatically by a ***special-purpose compiler***, called `genfft`, which can only produce optimized Fourier transform programs, but it excels at this task. `genfft` separates the logic of an algorithm from its implementation. The user specifies an algorithm at a high level (the "program"), and also how he or she wants the code to be implemented (the "metaprogram"). The advantage of metaprogramming is twofold. First, `genfft` is necessary to produce a space of decompositions large enough for self-optimization to be effective, since it would be impractical to write all codelets by hand. For example, the current FFTW system comprises 120 codelets for a total of more than 56,000 lines of code. Only a few codelets are used in typical situations, but it is important that all be available in order to be able to select the fast ones. Second, the distinction between the program and the metaprogram allows for easy changes in case we are desperate because every other portability technique fails. For example, `genfft` was at one point modified to generate code for processors, such as the PowerPC [83], which feature a fused multiply-add instruction. (This instruction computes $a \leftarrow a + bc$ in one cycle.) This modification required only 30 lines of code, and it improved the performance of FFTW on the PowerPC by 5-10%, although it was subsequently disabled because it slowed down FFTW on other machines. This example shows that machine-specific optimizations can be easily implemented if necessary. While less desirable than a fully automatic system, changing 30 lines is still better than changing 56,000.

While recursive divide and conquer algorithms suffer from the overheads of procedure calls, `genfft` helps overcoming the performance costs of the recursion. Codelets incur no recursion overhead in the codelets, because `genfft` unrolls the recursion completely. The main FFTW self-optimizing algorithm is also explicitly recursive, and it calls a codelet at the leaf of the recursion. Since codelets perform a significant amount of work, however, the overhead of this recursion is negligible. The FFTW system is described in Chapter 6.

> *This [other algorithm for univariate minimization] is not so nice. It took 17 iterations [where Brent's algorithm took 5] and we didn't get anywhere near as good an answer as we got with Brent. On the other hand, we understand how this works!*
>
> (Gerald Jay Sussman)

1.2 The methods of this dissertation

Our discussion of portable high performance draws ideas and methods from both the computer theory and systems literatures. In some cases our discussion will be entirely theoretical, like for example the asymptotic analysis of cache-oblivious algorithms. As is customary in theoretical analyses, we assume an idealized model and we happily disregard constant factors. In other cases, we will discuss at length implementation details whose only purpose is to save a handful CPU cycles. The Cilk work-stealing protocol is an example of this systems approach. You should not be surprised if we use these complementary sets of techniques, because the nature of the problem of portable high performance demands both. Certainly, we cannot say that a technique is high-performance if it has not been implemented, and therefore in this dissertation we pay attention to many implementation details and to empirical performance results. On the other hand, we cannot say anything about the portability of a technique unless we prove mathematically that the technique works on all machines. Consequently, this dissertation oscillates between theory and practice, aiming at understanding systems and algorithms from both points of view whenever possible, and you should be prepared to switch mind set from time to time.

1.3 Contributions

This dissertation shows how to write fast programs whose performance is portable. My main contributions consist in two portable high-performance software systems, and in theoretical analyses of portable high-performance algorithms and systems.

- *The Cilk language and an efficient implementation of Cilk on SMP's.* Cilk provides simple yet powerful constructs for expressing parallelism in an application. The language provides the programmer with parallel semantics that are easy to understand and use. Cilk's compilation and runtime strategies, which are inspired by the "work-first principle," are effective for writing portable high-performance parallel programs.

- *Cache-oblivious algorithms* provide performance and portability across platforms with different cache sizes. They are oblivious to the parameters of the memory hierarchy, and yet they use multiple levels of caches asymptotically optimally. This document presents cache-oblivious algorithms for matrix transpose and multiplication, FFT, and sorting that are asymptotically as good as previously known cache-aware algorithms, and provably optimal for those problems whose optimal cache complexity is known.

- *The location consistency memory model and the* BACKER *coherence algorithm* marry Cilk with cache-oblivious algorithms. This document proves good performance bounds for Cilk programs that uses location consistency.

- *The FFTW self-optimizing library* implements Fourier transforms of complex and real data in one or more dimensions. While FFTW does not require machine-specific performance tuning, its performance is comparable with or better than codes that were tuned for specific machines.

The rest of this dissertation is organized as follows. Chapter 2 describes the work-first principle and the implementation of Cilk-5. Chapter 3 defines cache obliviousness and gives cache-oblivious algorithms for matrix transpose, multiplication, FFT, and sorting. Chapter 4 presents location consistency and BACKER, and analyzes the performance of Cilk programs that use hierarchical shared memory. Chapter 5 presents the computation-centric theory of memory models. Chapter 6 describes the FFTW self-optimizing library and `genfft`. Finally, Chapter 7 offers some concluding remarks.

Chapter 2

Cilk

This chapter describes the ***Cilk*** system, which copes with parallelism in portable high-performance programs. Portability in the context of parallelism is usually called ***scalability***: a program scales if it attains good parallel speed-up. To really attain portable parallel high performance, however, we must write parallel programs that both "scale up" and "scale down" to run efficiently on a single processor—as efficiently as any sequential program that performs the same task. In this way, users can exploit whatever hardware is available, and developers do not need to maintain separate sequential and parallel versions of the same code.

Cilk is a multithreaded language for parallel programming that generalizes the semantics of C by introducing simple linguistic constructs for parallel control. The Cilk language implemented by the Cilk-5 release [38] uses the theoretically efficient scheduler from [25], but it was designed to scale down as well as to scale up. Typically, a Cilk program runs on a single processor with less than 5% slowdown relatively to a comparable C program. Cilk-5 is designed to run efficiently on contemporary symmetric multiprocessors (SMP's), which provide hardware support for shared memory. The Cilk group has coded many applications in Cilk, including the ⋆Socrates and Cilkchess chess-playing programs which have won prizes in international competitions. I was part of the team of Cilk programmers which won First Prize, undefeated in all matches, in the ICFP'98 Programming Contest sponsored by the 1998 International Conference on Functional Programming.[1]

Cilk's constructs for parallelism are simple. Parallelism in Cilk is expressed with call/return semantics, and the language has a simple "inlet" mechanism for nondeterministic control. The philosophy behind Cilk development has been to make the Cilk language a true parallel extension of C, both semantically and with respect to performance. On a parallel computer, Cilk control constructs allow the program to execute in parallel. If the Cilk keywords for parallel control are elided from a Cilk program, however, a syntactically and semantically correct C program results,

This chapter represents joint work with Charles Leiserson and Keith Randall. A preliminary version appears in [58].

[1] Cilk is not a functional language, but the contest was open to entries in any programming language.

which we call the ***C elision*** (or more generally, the ***serial elision***) of the Cilk program. Cilk is a ***faithful*** extension of C, because the C elision of a Cilk program is a correct implementation of the semantics of the program. On one processor, a parallel Cilk program scales down to run nearly as fast as its C elision.

Unlike in Cilk-1 [29], where the Cilk scheduler was an identifiable piece of code, in Cilk-5 both the compiler and runtime system bear the responsibility for scheduling. To obtain efficiency, we have, of course, attempted to reduce scheduling overheads. Some overheads have a larger impact on execution time than others, however. The framework for identifying and optimizing the common cases is provided by a theoretical understanding of Cilk's scheduling algorithm [25, 30]. According to this abstract theory, the performance of a Cilk computation can be characterized by two quantities: its ***work***, which is the total time needed to execute the computation serially, and its ***critical-path length***, which is its execution time on an infinite number of processors. (Cilk provides instrumentation that allows a user to measure these two quantities.) Within Cilk's scheduler, we can identify a given cost as contributing to either work overhead or critical-path overhead. Much of the efficiency of Cilk derives from the following principle, which will be justified in Section 2.3.

> ***The work-first principle:*** *Minimize the scheduling overhead borne by the work of a computation. Specifically, move overheads out of the work and onto the critical path.*

The work-first principle was used informally during the design of earlier Cilk systems, but Cilk-5 exploited the principle explicitly so as to achieve high performance. The work-first principle inspired a "two-clone" strategy for compiling Cilk programs. The `cilk2c` compiler [111] is a type-checking, source-to-source translator that transforms a Cilk source into a C postsource which makes calls to Cilk's runtime library. The C postsource is then run through the `gcc` compiler to produce object code. The `cilk2c` compiler produces two clones of every Cilk procedure—a "fast" clone and a "slow" clone. The fast clone, which is identical in most respects to the C elision of the Cilk program, executes in the common case where serial semantics suffice. The slow clone is executed in the infrequent case when parallel semantics and its concomitant bookkeeping are required. All communication due to scheduling occurs in the slow clone and contributes to critical-path overhead, but not to work overhead.

The work-first principle also inspired a Dijkstra-like [46], shared-memory, mutual-exclusion protocol as part of the runtime load-balancing scheduler. Cilk's scheduler uses a "work-stealing" algorithm in which idle processors, called ***thieves***, "steal" threads from busy processors, called ***victims***. Cilk's scheduler guarantees that the cost of stealing contributes only to critical-path overhead, and not to work overhead. Nevertheless, it is hard to avoid the mutual-exclusion costs incurred by a potential victim, which contribute to work overhead. To minimize work overhead, instead of using locking, Cilk's runtime system uses a Dijkstra-like protocol (which we call the ***THE***) protocol, to manage the runtime deque of ready threads in the work-stealing algorithm. An added advantage of the THE protocol is that it allows an exception to be signaled to a working processor with no additional work overhead, a feature used in Cilk's abort mechanism.

Cilk features a provably efficient scheduler, but it cannot magically make sequential programs parallel. To write portable parallel high performance, we must design scalable algorithms. In this chapter, we will give simple examples of parallel divide-and-conquer Cilk algorithms for matrix multiplication and sorting, and we will learn how to analyze work and critical-path length of Cilk algorithms. The combination of these analytic techniques with the efficiency of the Cilk scheduler allows us to write portable high-performance programs that cope with parallelism effectively.

The remainder of this chapter is organized as follows. Section 2.1 summarizes the development history of Cilk. Section 2.2 overviews the basic features of the Cilk language. Section 2.3 justifies the work-first principle. Section 2.4 analyzes the work and critical-path length of example Cilk algorithms. Section 2.5 describes how the two-clone strategy is implemented, and Section 2.6 presents the THE protocol. Section 2.7 gives empirical evidence that the Cilk-5 scheduler is efficient. Section 2.8 presents related work.

2.1 History of Cilk

While the following sections describe Cilk-5 as it is today, it is important to start with a brief summary of Cilk's history, so that you can learn how the system evolved to its current state.

The original 1994 Cilk-1 release [25, 29, 85] featured the provably efficient, randomized, "work-stealing" scheduler by Blumofe and Leiserson [25, 30]. The Cilk-1 language was clumsy and hard to program, however, because parallelism was exposed "by hand" using explicit continuation passing. Nonetheless, the ⋆Socrates chess program was written in this language, and it placed 3rd in the 1994 International Computer Chess Championship running on NCSA's 512-node CM5.

I became involved in the development of Cilk starting with Cilk-2. This system introduced the same call/return semantics that Cilk-5 uses today. This innovation was made possible by the outstanding work done by Rob Miller [111] on the `cilk2c` type-checking preprocessor. As the name suggests, `cilk2c` translates Cilk into C, performing semantic and dataflow analysis in the process. Most of Rob's `cilk2c` is still used in the current Cilk-5.

Cilk-3 added shared memory to Cilk. The innovation of Cilk-3 consisted in a novel memory model called ***dag consistency*** [27, 26] and of the BACKER coherence algorithm to support it. Cilk-3 was an evolutionary dead end as far as Cilk is concerned, because it implemented shared memory in software using special keywords to denote shared variables, and both these techniques disappeared from later versions of Cilk. The system was influential, however, in shaping the way the Cilk authors thought about shared memory and multithreaded algorithms. Dag consistency led to the computation-centric theory of memory models described in Chapter 5. The analysis of dag-consistent algorithms of [26] led to the notion of cache obliviousness, which is described in

Chapter 3. Finally, the general algorithmic framework of Cilk and of cache-oblivious algorithms provided a design model for FFTW (see Chapter 6).

While the first three Cilk systems were primarily developed on MPP's such as the Thinking Machines CM-5, the Cilk-4 system was targeted at symmetric multiprocessors. The system was based on a novel "two-clone" compilation strategy (see Section 2.5 and [58]) that Keith Randall invented. The Cilk language itself evolved to support "inlets" and nondeterministic programs. (See Section 2.2.) Cilk-4 was designed at the beginning of 1996 and written in the spring. The new implementation was made possible by a substantial and unexpected donation of SMP machines by Sun Microsystems.

It soon became apparent, however, that the Cilk-4 system was too complicated, and in the Fall of 1996 I decided to experiment with my own little Cilk system (initially called Milk, then Cilk-5). Cilk-4 managed virtual memory explicitly in order to maintain the illusion of a cactus stack [113], but this design decision turned out to be a mistake, because the need of maintaining a shared page table complicated the implementation enormously, and memory mapping from user space is generally slow in current operating systems.[2] The new Cilk-5 runtime system was engineered from scratch with simplicity as primary goal, and it used a simple heap-based memory manager. The `cilk2c` compiler did not change at all. While marginally slower than Cilk-4 on one processor, Cilk-5 turned out to be faster on multiple processors because of simpler protocols and fewer interactions with the operating system. In addition to this new runtime system, Cilk-5 featured a new debugging tool called the "Nondeterminator" [52, 37], which finds data races in Cilk programs.

2.2 The Cilk language

This section presents a brief overview of the Cilk extensions to C as supported by Cilk-5. (For a complete description, consult the Cilk-5 manual [38].) The key features of the language are the specification of parallelism and synchronization, through the `spawn` and `sync` keywords, and the specification of nondeterminism, using `inlet` and `abort`.

The basic Cilk language can be understood from an example. Figure 2-1 shows a Cilk program that computes the nth Fibonacci number.[3] Observe that the program would be an ordinary C program if the three keywords `cilk`, `spawn`, and `sync` were elided.

The keyword `cilk` identifies `fib` as a ***Cilk procedure***, which is the parallel analog to a C function. Parallelism is created when the keyword `spawn` precedes the invocation of a procedure. The semantics of a spawn differs from a C function call only in that the parent can continue to execute in parallel with the child, instead of waiting for the child to complete as is done in C. Cilk's

[2]We could have avoid this mistake had we read Appel and Shao [13].

[3]This program uses an inefficient algorithm which runs in exponential time. Although logarithmic-time methods are known [42, p. 850], this program nevertheless provides a good didactic example.

```
#include <stdlib.h>
#include <stdio.h>
#include <cilk.h>

cilk int fib (int n)
{
    if (n<2) return n;
    else {
        int x, y;
        x = spawn fib (n-1);
        y = spawn fib (n-2);
        sync;
        return (x+y);
    }
}

cilk int main (int argc, char *argv[])
{
    int n, result;
    n = atoi(argv[1]);
    result = spawn fib(n);
    sync;
    printf ("Result: %d\n", result);
    return 0;
}
```

Figure 2-1: A simple Cilk program to compute the nth Fibonacci number in parallel (using a very bad algorithm).

scheduler takes the responsibility of scheduling the spawned procedures on the processors of the parallel computer.

A Cilk procedure cannot safely use the values returned by its children until it executes a `sync` statement. The `sync` statement is a local "barrier," not a global one as, for example, is used in message-passing programming environments such as MPI [134]. In the Fibonacci example, a `sync` statement is required before the statement `return (x+y)` to avoid the incorrect result that would occur if `x` and `y` are summed before they are computed. In addition to explicit synchronization provided by the `sync` statement, every Cilk procedure syncs implicitly before it returns, thus ensuring that all of its children terminate before it does.

Cactus stack. Cilk extends the semantics of C by supporting cactus stack [78, 113, 137] semantics for stack-allocated objects. From the point of view of a single Cilk procedure, a cactus stack behaves much like an ordinary stack. The procedure can allocate and free memory by incrementing and decrementing a stack pointer. The procedure views the stack as a linearly addressed space extending

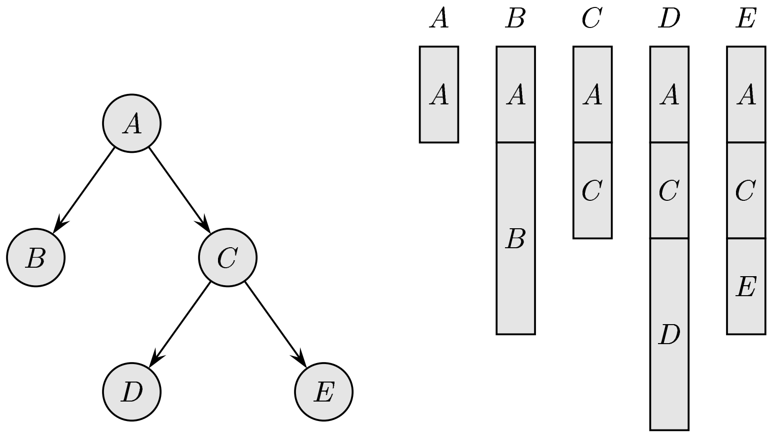


Figure 2-2: A cactus stack. The left-hand side shows a tree of procedures, where procedure A spawns procedures B and C, and procedure C spawns procedures D and E. The right-hand side shows the stack view for the 5 procedures. For examples, D "sees" the frames of procedures A and C, but not that of B.

back from its own stack frame to the frame of its parent and continuing to more distant ancestors. The stack becomes a cactus stack when multiple procedures execute in parallel, each with its own view of the stack that corresponds to its call history, as shown in Figure 2-2.

Cactus-stack allocation mirrors the advantages of an ordinary procedure stack. Procedure-local variables and arrays can be allocated and deallocated automatically by the runtime system in a natural fashion. Separate branches of the cactus stack are insulated from each other, allowing two threads to allocate and free objects independently, even though objects may be allocated with the same address. Procedures can reference common data through the shared portion of their stack address space.

Cactus stacks have many of the same limitations as ordinary procedure stacks [113]. For instance, a child thread cannot return to its parent a pointer to an object that it has allocated. Similarly, sibling procedures cannot share storage that they create on the stack. Just as with a procedure stack, pointers to objects allocated on the cactus stack can only be safely passed to procedures below the allocation point in the call tree. To alleviate these limitations, Cilk offers a heap allocator in the style of `malloc`/`free`.

Inlets. Ordinarily, when a spawned procedure returns, the returned value is simply stored into a variable in its parent's frame:

```
x = spawn foo(y);
```

Occasionally, one would like to incorporate the returned value into the parent's frame in a more complex way. Cilk provides an ***inlet*** feature for this purpose, which was inspired in part by the inlet feature of TAM [45].

```
cilk int fib (int n)
{
    int x = 0;
    inlet void summer (int result)
    {
        x += result;
        return;
    }

    if (n<2) return n;
    else {
        summer(spawn fib (n-1));
        summer(spawn fib (n-2));
        sync;
        return (x);
    }
}
```

Figure 2-3: Using an inlet to compute the nth Fibonacci number.

An inlet is essentially a C function internal to a Cilk procedure. In the normal syntax of Cilk, the spawning of a procedure must occur as a separate statement and not in an expression. An exception is made to this rule if the spawn is performed as an argument to an inlet call. In this case, the procedure is spawned, and when it returns, the inlet is invoked. In the meantime, control of the parent procedure proceeds to the statement following the inlet call. In principle, inlets can take multiple spawned arguments, but Cilk-5 has the restriction that exactly one argument to an inlet may be spawned and that this argument must be the first argument. If necessary, this restriction is easy to program around.

Figure 2-3 illustrates how the `fib()` function might be coded using inlets. The inlet `summer()` is defined to take a returned value `result` and add it to the variable `x` in the frame of the procedure that does the spawning. All the variables of `fib()` are available within `summer()`, since it is an internal function of `fib()`.[4]

No lock is required around the accesses to `x` by `summer`, because Cilk provides atomicity implicitly. The concern is that the two updates might occur in parallel, and if atomicity is not imposed, an update might be lost. Cilk provides implicit atomicity among the "threads" of a procedure instance, where a ***thread*** is a maximal sequence of instructions that does not contain a `spawn`, `sync`, or `return` (either explicit or implicit) statement. An inlet is precluded from containing `spawn` and `sync` statements, and thus it operates atomically as a single thread. Implicit atomicity simplifies

[4]The C elision of a Cilk program with inlets is not ANSI C, because ANSI C does not support internal C functions. Cilk is based on GNU C technology, however, which does provide this support.

reasoning about concurrency and nondeterminism without requiring locking, declaration of critical regions, and the like.

Cilk provides syntactic sugar to produce certain commonly used inlets implicitly. For example, the statement `x += spawn fib(n-1)` conceptually generates an inlet similar to the one in Figure 2-3.

Abort. Sometimes, a procedure spawns off parallel work which it later discovers is unnecessary. This "speculative" work can be aborted in Cilk using the `abort` primitive inside an inlet. A common use of `abort` occurs during a parallel search, where many possibilities are searched in parallel. As soon as a solution is found by one of the searches, one wishes to abort any currently executing searches as soon as possible so as not to waste processor resources. The `abort` statement, when executed inside an inlet, causes all of the already-spawned children of the procedure to terminate.

We considered using "futures" [76] with implicit synchronization, as well as synchronizing on specific variables, instead of using the simple `spawn` and `sync` statements. We realized from the work-first principle, however, that different synchronization mechanisms could have an impact only on the critical-path of a computation, and so this issue was of secondary concern. Consequently, we opted for implementation simplicity. Also, in systems that support relaxed memory-consistency models, the explicit `sync` statement can be used to ensure that all side-effects from previously spawned subprocedures have occurred.

In addition to the control synchronization provided by `sync`, Cilk programmers can use explicit locking to synchronize accesses to data, providing mutual exclusion and atomicity. Data synchronization is an overhead borne on the work, however, and although we have striven to minimize these overheads, fine-grain locking on contemporary processors is expensive. We are currently investigating how to incorporate atomicity into the Cilk language so that protocol issues involved in locking can be avoided at the user level. To aid in the debugging of Cilk programs that use locks, the Cilk group has developed a tool called the "Nondeterminator" [37, 52], which detects common synchronization bugs called ***data races***.

2.3 The work-first principle

This section justifies the work-first principle stated at the beginning of this chapter by showing that it follows from three assumptions. First, we assume that Cilk's scheduler operates in practice according to the theoretical analysis presented in [25, 30]. Second, we assume that in the common case, ample "parallel slackness" [145] exists, that is, the parallelism of a Cilk program exceeds the number of processors on which we run it by a sufficient margin. Third, we assume (as is indeed the case) that every Cilk program has a C elision against which its one-processor performance can be measured.

The theoretical analysis presented in [25, 30] cites two fundamental lower bounds as to how fast a Cilk program can run. Let us denote by T_P the execution time of a given computation on P processors. The work of the computation is then T_1 and its critical-path length is T_∞. For a computation with T_1 work, the lower bound $T_P \geq T_1/P$ must hold, because at most P units of work can be executed in a single step. In addition, the lower bound $T_P \geq T_\infty$ must hold, since a finite number of processors cannot execute faster than an infinite number.[5]

Cilk's randomized work-stealing scheduler [25, 30] executes a Cilk computation on P processors in expected time

$$T_P = T_1/P + O(T_\infty) , \tag{2.1}$$

assuming an ideal parallel computer. This equation resembles "Brent's theorem" [32, 71] and is optimal to within a constant factor, since T_1/P and T_∞ are both lower bounds. We call the first term on the right-hand side of Equation (2.1) the ***work*** term and the second term the ***critical-path*** term. Importantly, all communication costs due to Cilk's scheduler are borne by the critical-path term, as are most of the other scheduling costs. To make these overheads explicit, we define the ***critical-path overhead*** to be the smallest constant c_∞ such that

$$T_P \leq T_1/P + c_\infty T_\infty . \tag{2.2}$$

The second assumption needed to justify the work-first principle focuses on the "common-case" regime in which a parallel program operates. Define the ***parallelism*** as $\overline{P} = T_1/T_\infty$, which corresponds to the maximum possible speedup that the application can obtain. Define also the ***parallel slackness*** [145] to be the ratio $\overline{P}/P$. The ***assumption of parallel slackness*** is that $\overline{P}/P \gg c_\infty$, which means that the number P of processors is much smaller than the parallelism $\overline{P}$. Under this assumption, it follows that $T_1/P \gg c_\infty T_\infty$, and hence from Inequality (2.2) that $T_P \approx T_1/P$, and we obtain linear speedup. The critical-path overhead c_∞ has little effect on performance when sufficient slackness exists, although it does determine how much slackness must exist to ensure linear speedup.

Whether substantial slackness exists in common applications is a matter of opinion and empiricism, but we suggest that slackness is the common case. The expressiveness of Cilk makes it easy to code applications with large amounts of parallelism. For modest-sized problems, many applications exhibit a parallelism of over 200, yielding substantial slackness on contemporary SMP's. Even on Sandia National Laboratory's Intel Paragon, which contains 1824 nodes, the $\star$Socrates chess program (coded in Cilk-1) ran in its linear-speedup regime during the 1995 ICCA World Computer

[5]This abstract model of execution time ignores real-life details, such as memory-hierarchy effects, but is nonetheless quite accurate [29].

Chess Championship (where it placed second in a field of 24). Section 2.7 describes a dozen other diverse applications which were run on an 8-processor SMP with considerable parallel slackness. The parallelism of these applications increases with problem size, thereby ensuring they will be portable to large machines.

The third assumption behind the work-first principle is that every Cilk program has a C elision against which its one-processor performance can be measured. Let us denote by T_{S} the running time of the C elision. Then, we define the ***work overhead*** by $c_1 = T_1/T_{\mathrm{S}}$. Incorporating critical-path and work overheads into Inequality (2.2) yields

$$\begin{aligned} T_P &\leq c_1 T_{\mathrm{S}}/P + c_\infty T_\infty \\ &\approx c_1 T_{\mathrm{S}}/P \,, \end{aligned} \tag{2.3}$$

since we assume parallel slackness.

We can now restate the work-first principle precisely. *Minimize c_1, even at the expense of a larger c_∞*, because c_1 has a more direct impact on performance. Adopting the work-first principle may adversely affect the ability of an application to scale up, however, if the critical-path overhead c_∞ is too large. But, as we shall see in Section 2.7, critical-path overhead is reasonably small in Cilk-5, and many applications can be coded with large amounts of parallelism.

The work-first principle pervades the Cilk-5 implementation. The work-stealing scheduler guarantees that with high probability, only $O(PT_\infty)$ steal (migration) attempts occur (that is, $O(T_\infty)$ on average per processor), all costs for which are borne on the critical path. Consequently, the scheduler for Cilk-5 postpones as much of the scheduling cost as possible to when work is being stolen, thereby removing it as a contributor to work overhead. This strategy of amortizing costs against steal attempts permeates virtually every decision made in the design of the scheduler.

2.4 Example Cilk algorithms

In this section, we give example Cilk algorithms for matrix multiplication and sorting, and analyze their work and critical-path length. The matrix multiplication algorithm multiplies two $n \times n$ matrices using $\Theta(n^3)$ work with critical-path length $\Theta(\lg^2 n)$. The sorting algorithm sorts an array of n elements using work $\Theta(n \lg n)$ with a critical-path length of $\Theta(\lg^3 n)$. The parallelism of these algorithms is ample ($\overline{P} = \Theta(n^3/\lg^2 n)$ and $\overline{P} = \Theta(n/\lg^2 n)$ respectively). Since Cilk executes a program efficiently whenever $P \ll \overline{P}$, these algorithms are thus good candidates for portable high performance. In this section, we focus on the theoretical analysis of these algorithms. We will see in Section 2.7 that they also perform well in practice.

We start with the `matrixmul` matrix multiplication algorithm from [27]. To multiply the $n \times n$ matrix A by similar matrix B, `matrixmul` divides each matrix into four $n/2 \times n/2$ submatrices and uses the identity

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$
$$= \begin{bmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{bmatrix} + \begin{bmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{bmatrix} .$$

The idea of `matrixmul` is to recursively compute the 8 products of the submatrices of A and B in parallel, and then add the subproducts together in pairs to form the result using recursive matrix addition. In the base case $n = 1$, `matrixmul` computes the product directly.

Figure 2-4 shows Cilk code for an implementation of `matrixmul` that multiplies two square matrices `A` and `B` yielding the output matrix `R`. The Cilk procedure `matrixmul` takes as arguments pointers to the first block in each matrix as well as a variable `n` denoting the size of any row or column of the matrices. As `matrixmul` executes, values are stored into `R`, as well as into a temporary matrix `tmp`.

Both the work and the critical-path length for `matrixmul` can be computed using recurrences. The work $T_1(n)$ to multiply $n \times n$ matrices satisfies the recurrence $T_1(n) = 8T_1(n/2) + \Theta(n^2)$, since addition of two matrices can be done using $O(n^2)$ computational work, and thus, $T_1(n) = \Theta(n^3)$. To derive a recurrence for the critical-path length $T_\infty(n)$, we observe that with an infinite number of processors, only one of the 8 submultiplications is the bottleneck, because the 8 multiplications can execute in parallel. Consequently, the critical-path length $T_\infty(n)$ satisfies $T_\infty(n) = T_\infty(n/2) + \Theta(\lg n)$, because the parallel addition can be accomplished recursively with a critical path of length $\Theta(\lg n)$. The solution to this recurrence is $T_\infty(n) = \Theta(\lg^2 n)$.

Algorithms exist for matrix multiplication with a shorter critical-path length. Specifically, two $n \times n$ matrices can be multiplied using $\Theta(n^3)$ work with a critical-path of $\Theta(\lg n)$ [98], which is shorter than `matrixmul`'s critical path. As we will see in Chapter 3, however, memory-hierarchy considerations play a role in addition to work and critical path in the design of portable high-performance algorithms. In Chapter 3 we will prove that `matrixmul` uses the memory hierarchy efficiently, and in fact we will argue that `matrixmul` should be the preferred way to code even a *sequential* program.

We now discuss the Cilksort parallel sorting algorithm, which is a variant of ordinary mergesort. Cilksort is inspired by [10]. Cilksort begins by dividing an array of elements into two halves, and it sorts each half recursively in parallel. It then merges the two sorted halves back together, but in a divide-and-conquer approach rather than with the usual serial merge. Say that we wish to merge sorted arrays A and B. Without loss of generality, assume that A is larger than B. We begin by dividing array A into two halves, letting A_1 denote the lower half and A_2 the upper. We then take the middle element of A and use a binary search to discover where that element should fit into array

```
cilk void matrixmul(int n, float *A,
                           float *B,
                           float *R)
 {
   if (n == 1)
     *R = *A * *B;
   else {
     float *A11,*A12,*A21,*A22,*B11,*B12,*B21,*B22;
     float *A11B11,*A11B12,*A21B11,*A21B12,
           *A12B21,*A12B22,*A22B21,*A22B22;
     float tmp[n*n];

     /* get pointers to input submatrices */
     partition(n, A, &A11, &A12, &A21, &A22);
     partition(n, B, &B11, &B12, &B21, &B22);

     /* get pointers to result submatrices */
     partition(n, R, &A11B11, &A11B12, &A21B11, &A21B12);
     partition(n, tmp, &A12B21, &A12B22, &A22B21, &A22B22);

     /* solve subproblems recursively */
     spawn matrixmul(n/2, A11, B11, A11B11);
     spawn matrixmul(n/2, A11, B12, A11B12);
     spawn matrixmul(n/2, A21, B12, A21B12);
     spawn matrixmul(n/2, A21, B11, A21B11);

     spawn matrixmul(n/2, A12, B21, A12B21);
     spawn matrixmul(n/2, A12, B22, A12B22);
     spawn matrixmul(n/2, A22, B22, A22B22);
     spawn matrixmul(n/2, A22, B21, A22B21);
     sync;

     /* add results together into R */
     spawn matrixadd(n, tmp, R);
     sync;
   }
   return;
 }
```

Figure 2-4: Cilk code for recursive matrix multiplication.

B. This search yields a division of array B into subarrays B_1 and B_2. We then recursively merge A_1 with B_1 and A_2 with B_2 in parallel and concatenate the results, which yields the desired fully merged version of A and B.

To analyze work and critical path of Cilksort, we first analyze the merge procedure. Let n be the total size of the two arrays A and B. The merge algorithm splits a problem of size n into two problems of size n_1 and n_2, where $n_1 + n_2 = n$ and $\max\{n_1, n_2\} \leq (3/4)n$, and it uses $O(\lg n)$ work for the binary search. The work recurrence is therefore $T_1(n) = T_1(n_1) + T_1(n_2) + O(\lg n)$, whose solution is $T_1(n) = \Theta(n)$. The critical path recurrence is given by $T_\infty(n) = T_\infty(\max\{n_1, n_2\}) + O(\lg n)$, because the two subproblems can be solved in parallel but they must both wait for the binary search to complete. Consequently, the critical path for merging is $T_\infty(n) = \Theta(\lg^2 n)$.

We now analyze Cilksort using the analysis of the merge procedure. Cilksort splits a problem of size n into two subproblems of size $n/2$, and merges the results. The work recurrence is $T_1(n) = 2T_1(n/2) + \Theta(n)$, where $\Theta(n)$ work derives from the merge procedure. Similarly, the critical path recurrence is $T_\infty(n) = T_\infty(n/2) + \Theta(\lg^2 n)$, where $\Theta(\lg^2 n)$ is the critical path of the merge step. We conclude that Cilksort has work $\Theta(n \lg n)$ and critical path $\Theta(\lg^3 n)$.

Cilksort is a simple algorithm that works well in practice. It uses optimal work, and its critical path is reasonably short. As we will see in Section 2.7, Cilksort is only about 20% slower than optimized sequential quicksort, and its parallelism is more than 1000 for n =4,100,000. Cilksort thus qualifies as a portable high-performance parallel algorithm. A drawback of Cilksort is that it does not use the memory hierarchy optimally. In Chapter 3 we will discuss more complicated sorting algorithms that are optimal in this sense.

2.5 Cilk's compilation strategy

This section describes how our `cilk2c` compiler generates C postsource from a Cilk program. As dictated by the work-first principle, our compiler and scheduler are designed to reduce the work overhead as much as possible. Our strategy is to generate two clones of each procedure—a ***fast*** clone and a ***slow*** clone. The fast clone operates much as does the C elision and has little support for parallelism. The slow clone has full support for parallelism, along with its concomitant overhead. In the rest of this section, we first describe the Cilk scheduling algorithm. Then, we describe how the compiler translates the Cilk language constructs into code for the fast and slow clones of each procedure. Lastly, we describe how the runtime system links together the actions of the fast and slow clones to produce a complete Cilk implementation. We can say, somewhat informally, that in Cilk the fast clone takes care of high-performance, since it runs with minimal overhead, while the slow clone takes care of portability, since it allows parallelism to be exploited.

As in lazy task creation [112], in Cilk-5 each processor (called a ***worker***) maintains a ***ready***

deque (doubly-ended queue) of ready procedures (technically, procedure instances). Each deque has two ends, a ***head*** and a ***tail***, from which procedures can be added or removed. A worker operates locally on the tail of its own deque, treating it much as C treats its call stack, pushing and popping spawned activation frames. When a worker runs out of work, it becomes a ***thief*** and attempts to steal a procedure another worker, called its ***victim***. The thief steals the procedure from the head of the victim's deque, the opposite end from which the victim is working.

When a procedure is spawned, the fast clone runs. Whenever a thief steals a procedure, however, the procedure is converted into a slow clone. The Cilk scheduler guarantees that the number of steals is small when sufficient slackness exists, and thus we expect the fast clones to be executed most of the time. Thus, the work-first principle reduces to minimizing costs in the fast clone, which contribute more heavily to work overhead. Minimizing costs in the slow clone, although a desirable goal, is less important, since these costs contribute less heavily to work overhead and more to critical-path overhead.

We minimize the costs of the fast clone by exploiting the structure of the Cilk scheduler. Because we convert a procedure to its slow clone when it is stolen, we maintain the invariant that a fast clone has never been stolen. Furthermore, none of the descendants of a fast clone have been stolen either, since the strategy of stealing from the heads of ready deques guarantees that parents are stolen before their children. As we will see, this simple fact allows many optimizations to be performed in the fast clone.

We now describe how our `cilk2c` compiler generates postsource C code for the `fib` procedure from Figure 2-1. An example of the postsource for the fast clone of `fib` is given in Figure 2-5. The generated C code has the same general structure as the C elision, with a few additional statements. In lines 4–5, an ***activation frame*** is allocated for `fib` and initialized. The Cilk runtime system uses activation frames to represent procedure instances. Using techniques similar to [72, 73], our inlined allocator typically takes only a few cycles. The frame is initialized in line 5 by storing a pointer to a static structure, called a signature, describing `fib`.

The first spawn in `fib` is translated into lines 12–18. In lines 12–13, the state of the `fib` procedure is saved into the activation frame. The saved state includes the program counter, encoded as an entry number, and all live, dirty variables. Then, the frame is pushed on the runtime deque in lines 14–15.[6] Next, we call the `fib` routine as we would in C. Because the `spawn` statement itself compiles directly to its C elision, the postsource can exploit the optimization capabilities of the C compiler, including its ability to pass arguments and receive return values in registers rather than in memory.

After `fib` returns, lines 17–18 check to see whether the parent procedure has been stolen. If it has, we return immediately with a dummy value. Since all of the ancestors have been stolen as

[6] If the shared memory is not sequentially consistent, a memory fence must be inserted between lines 14 and 15 to ensure that the surrounding writes are executed in the proper order.

```
int fib (int n)
{
    fib_frame *f;                   frame pointer
    f = alloc(sizeof(*f));          allocate frame
    f->sig = fib_sig;               initialize frame
    if (n<2) {
        free(f, sizeof(*f));        free frame
        return n;
    }
    else {
        int x, y;
        f->entry = 1;               save PC
        f->n = n;                   save live vars
        *T = f;                     store frame pointer
        push();                     push frame
        x = fib (n-1);              do C call
        if (pop(x) == FAILURE)      pop frame
            return 0;               frame stolen
        ...                         second spawn
        ;                           sync is free!
        free(f, sizeof(*f));        free frame
        return (x+y);
    }
}
```

Figure 2-5: The fast clone generated by `cilk2c` for the `fib` procedure from Figure 2-1. The code for the second spawn is omitted. The functions `alloc` and `free` are inlined calls to the runtime system's fast memory allocator. The signature `fib_sig` contains a description of the `fib` procedure, including a pointer to the slow clone. The `push` and `pop` calls are operations on the scheduling deque and are described in detail in Section 2.6.

well, the C stack quickly unwinds and control is returned to the runtime system.[7] The protocol to check whether the parent procedure has been stolen is quite subtle—we postpone discussion of its implementation to Section 2.6. If the parent procedure has not been stolen, it continues to execute at line 19, performing the second spawn, which is not shown.

In the fast clone, all `sync` statements compile to no-ops. Because a fast clone never has any children when it is executing, we know at compile time that all previously spawned procedures have completed. Thus, no operations are required for a `sync` statement, as it always succeeds. For example, line 20 in Figure 2-5, the translation of the `sync` statement is just the empty statement. Finally, in lines 21–22, `fib` deallocates the activation frame and returns the computed result to its parent procedure.

The slow clone is similar to the fast clone except that it provides support for parallel execution. When a procedure is stolen, control has been suspended between two of the procedure's threads, that is, at a spawn or sync point. When the slow clone is resumed, it uses a `goto` statement to restore the program counter, and then it restores local variable state from the activation frame. A `spawn` statement is translated in the slow clone just as in the fast clone. For a `sync` statement, `cilk2c` inserts a call to the runtime system, which checks to see whether the procedure has any spawned children that have not returned. Although the parallel bookkeeping in a slow clone is substantial, it contributes little to work overhead, since slow clones are rarely executed.

The separation between fast clones and slow clones also allows us to compile inlets and abort statements efficiently in the fast clone. An inlet call compiles as efficiently as an ordinary spawn. For example, the code for the inlet call from Figure 2-3 compiles similarly to the following Cilk code:

```
tmp = spawn fib(n-1);
summer(tmp);
```

Implicit inlet calls, such as `x += spawn fib(n-1)`, compile directly to their C elisions. An `abort` statement compiles to a no-op just as a `sync` statement does, because while it is executing, a fast clone has no children to abort.

The runtime system provides the glue between the fast and slow clones that makes the whole system work. It includes protocols for stealing procedures, returning values between processors, executing inlets, aborting computation subtrees, and the like. All of the costs of these protocols can be amortized against the critical path, so their overhead does not significantly affect the running time when sufficient parallel slackness exists. The portion of the stealing protocol executed by the worker contributes to work overhead, however, thereby warranting a careful implementation. We discuss this protocol in detail in Section 2.6.

[7] The `setjmp`/`longjmp` facility of C could have been used as well, but our unwinding strategy is simpler.

The work overhead of a `spawn` in Cilk-5 is only a few reads and writes in the fast clone—3 reads and 5 writes for the `fib` example. We will experimentally quantify the work overhead in Section 2.7. Some work overheads still remain in our implementation, however, including the allocation and freeing of activation frames, saving state before a spawn, pushing and popping of the frame on the deque, and checking if a procedure has been stolen. A portion of this work overhead is due to the fact that Cilk-5 is duplicating the work the C compiler performs, but as Section 2.7 shows, this overhead is small. Although a production Cilk compiler might be able eliminate this unnecessary work, it would likely compromise portability.

In Cilk-4, the precursor to Cilk-5, we took the work-first principle to the extreme. Cilk-4 performed stack-based allocation of activation frames, since the work overhead of stack allocation is smaller than the overhead of heap allocation. Because of the "cactus stack" [113] semantics of the Cilk stack,[8] however, Cilk-4 had to manage the virtual-memory map on each processor explicitly, as was done in [137]. The work overhead in Cilk-4 for frame allocation was little more than that of incrementing the stack pointer, but whenever the stack pointer overflowed a page, an expensive user-level interrupt ensued, during which Cilk-4 would modify the memory map. Unfortunately, the operating-system mechanisms supporting these operations were too slow and unpredictable, and the possibility of a page fault in critical sections led to complicated protocols. Even though these overheads could be charged to the critical-path term, in practice, they became so large that the critical-path term contributed significantly to the running time, thereby violating the assumption of parallel slackness. A one-processor execution of a program was indeed fast, but insufficient slackness sometimes resulted in poor parallel performance.

In Cilk-5, we simplified the allocation of activation frames by simply using a heap. In the common case, a frame is allocated by removing it from a free list. Deallocation is performed by inserting the frame into the free list. No user-level management of virtual memory is required, except for the initial setup of shared memory. Heap allocation contributes only slightly more than stack allocation to the work overhead, but it saves substantially on the critical path term. On the downside, heap allocation can potentially waste more memory than stack allocation due to fragmentation. For a careful analysis of the relative merits of stack and heap based allocation that supports heap allocation, see the paper by Appel and Shao [13]. For an equally careful analysis that supports stack allocation, see [110].

Thus, although the work-first principle gives a general understanding of where overheads should be borne, our experience with Cilk-4 showed that large enough critical-path overheads can tip the scales to the point where the assumptions underlying the principle no longer hold. We believe that Cilk-5 work overhead is nearly as low as possible, given our goal of generating portable C output

[8] Suppose a procedure A spawns two children B and C. The two children can reference objects in A's activation frame, but B and C do not see each other's frame.

from our compiler.[9] Other researchers have been able to reduce overheads even more, however, at the expense of portability. For example, lazy threads [68] obtains efficiency at the expense of implementing its own calling conventions, stack layouts, etc. Although we could in principle incorporate such machine-dependent techniques into our compiler, we feel that Cilk-5 strikes a good balance between performance and portability. We also feel that the current overheads are sufficiently low that other problems, notably minimizing overheads for data synchronization, deserve more attention.

2.6 Implementation of work-stealing

In this section, we describe Cilk-5's work-stealing mechanism, which is based on a Dijkstra-like [46], shared-memory, mutual-exclusion protocol called the "THE" protocol. In accordance with the work-first principle, this protocol has been designed to minimize work overhead. For example, on a 167-megahertz UltraSPARC I, the `fib` program with the THE protocol runs about 25% faster than with hardware locking primitives. We first present a simplified version of the protocol. Then, we discuss the actual implementation, which allows exceptions to be signaled with no additional overhead.

Several straightforward mechanisms might be considered to implement a work-stealing protocol. For example, a thief might interrupt a worker and demand attention from this victim. This strategy presents problems for two reasons. First, the mechanisms for signaling interrupts are slow, and although an interrupt would be borne on the critical path, its large cost could threaten the assumption of parallel slackness. Second, the worker would necessarily incur some overhead on the work term to ensure that it could be safely interrupted in a critical section. As an alternative to sending interrupts, thieves could post steal requests, and workers could periodically poll for them. Once again, however, a cost accrues to the work overhead, this time for polling. Techniques are known that can limit the overhead of polling [50], but they require the support of a sophisticated compiler.

The work-first principle suggests that it is reasonable to put substantial effort into minimizing work overhead in the work-stealing protocol. Since Cilk-5 is designed for shared-memory machines, we chose to implement work-stealing through shared-memory, rather than with message-passing, as might otherwise be appropriate for a distributed-memory implementation. In our implementation, both victim and thief operate directly through shared memory on the victim's ready deque. The crucial issue is how to resolve the race condition that arises when a thief tries to steal the same frame that its victim is attempting to pop. One simple solution is to add a lock to the deque using relatively heavyweight hardware primitives like Compare-And-Swap or Test-And-Set. Whenever a thief or worker wishes to remove a frame from the deque, it first grabs the lock. This

[9] Although the runtime system requires some effort to port between architectures, the compiler requires no changes whatsoever for different platforms.

solution has the same fundamental problem as the interrupt and polling mechanisms just described, however. Whenever a worker pops a frame, it pays the heavy price to grab a lock, which contributes to work overhead.

Consequently, we adopted a solution that employs Dijkstra's protocol for mutual exclusion [46], which assumes only that reads and writes are atomic. Because our protocol uses three atomic shared variables `T`, `H`, and `E`, we call it the ***THE*** protocol. The key idea is that actions by the worker on the tail of the queue contribute to work overhead, while actions by thieves on the head of the queue contribute only to critical-path overhead. Therefore, in accordance with the work-first principle, we attempt to move costs from the worker to the thief. To arbitrate among different thieves attempting to steal from the same victim, we use a hardware lock, since this overhead can be amortized against the critical path. To resolve conflicts between a worker and the sole thief holding the lock, however, we use a lightweight Dijkstra-like protocol which contributes minimally to work overhead. A worker resorts to a heavyweight hardware lock only when it encounters an actual conflict with a thief, in which case we can charge the overhead that the victim incurs to the critical path.

In the rest of this section, we describe the THE protocol in detail. We first present a simplified protocol that uses only two shared variables `T` and `H` designating the tail and the head of the deque, respectively. Later, we extend the protocol with a third variable `E` that allows exceptions to be signaled to a worker. The exception mechanism is used to implement Cilk's `abort` statement. Interestingly, this extension does not introduce any additional work overhead.

The pseudocode of the simplified THE protocol is shown in Figure 2-6. Assume that shared memory is sequentially consistent [96].[10] The code assumes that the ready deque is implemented as an array of frames. The head and tail of the deque are determined by two indices `T` and `H`, which are stored in shared memory and are visible to all processors. The index `T` points to the first unused element in the array, and `H` points to the first frame on the deque. Indices grow from the head towards the tail so that under normal conditions, we have $\mathtt{T} \geq \mathtt{H}$. Moreover, each deque has a lock `L` implemented with atomic hardware primitives or with OS calls.

The worker uses the deque as a stack. (See Section 2.5.) Before a `spawn`, it pushes a frame onto the tail of the deque. After a `spawn`, it pops the frame, unless the frame has been stolen. A thief attempts to steal the frame at the head of the deque. Only one thief at the time may steal from the deque, since a thief grabs `L` as its first action. As can be seen from the code, the worker alters `T` but not `H`, whereas the thief only increments `H` and does not alter `T`.

The only possible interaction between a thief and its victim occurs when the thief is incrementing `H` while the victim is decrementing `T`. Consequently, it is always safe for a worker to append a new frame at the end of the deque (`push`) without worrying about the actions of the thief. For a

[10] If the shared memory is not sequentially consistent, a memory fence must be inserted between lines 5 and 6 of the worker/victim code and between lines 3 and 4 of the thief code to ensure that these instructions are executed in the proper order.

Worker/Victim

```
push() {
  T++;
}

pop() {
  T--;
  if (H > T) {
    T++;
    lock(L);
    T--;
    if (H > T) {
      T++;
      unlock(L);
      return FAILURE;
    }
    unlock(L);
  }
  return SUCCESS;
}
```

Thief

```
steal() {
  lock(L);
  H++;
  if (H > T) {
    H--;
    unlock(L);
    return FAILURE;
  }
  unlock(L);
  return SUCCESS;
}
```

Figure 2-6: Pseudocode of a simplified version of the THE protocol. The left part of the figure shows the actions performed by the victim, and the right part shows the actions of the thief. None of the actions besides reads and writes are assumed to be atomic. For example, `T--;` can be implemented as `tmp = T; tmp = tmp - 1; T = tmp;`.

pop operation, there are three cases, which are shown in Figure 2-7. In case (a), the thief and the victim can both obtain a frame from the deque. In case (b), the deque contains only one frame. If the victim decrements T without interference from thieves, it gets the frame. Similarly, a thief can steal the frame as long as its victim is not trying to obtain it. If both the thief and the victim try to grab the frame, however, the protocol guarantees that at least one of them discovers that $H > T$. If the thief discovers that $H > T$, it restores H to its original value and retreats. If the victim discovers that $H > T$, it restores T to its original value and restarts the protocol after having acquired L. With L acquired, no thief can steal from this deque so the victim can pop the frame without interference (if the frame is still there). Finally, in case (c) the deque is empty. If a thief tries to steal, it will always fail. If the victim tries to pop, the attempt fails and control returns to the Cilk runtime system. The protocol cannot deadlock, because each process holds only one lock at a time.

We now argue that the THE protocol contributes little to the work overhead. Pushing a frame involves no overhead beyond updating T. In the common case where a worker can successfully pop a frame, the pop protocol performs only 6 operations—2 memory loads, 1 memory store, 1 decrement, 1 comparison, and 1 (predictable) conditional branch. Moreover, in the common case where no thief operates on the deque, both H and T can be cached exclusively by the worker. The expensive operation of a worker grabbing the lock L occurs only when a thief is simultaneously

Figure 2-7: The three cases of the ready deque in the simplified THE protocol. A dark entry indicates the presence of a frame at a certain position in the deque. The head and the tail are marked by `T` and `H`.

trying to steal the frame being popped. Since the number of steal attempts depends on T_∞, not on T_1, the relatively heavy cost of a victim grabbing `L` can be considered as part of the critical-path overhead c_∞ and does not influence the work overhead c_1.

We ran some experiments to determine the relative performance of the THE protocol versus the straightforward protocol in which `pop` just locks the deque before accessing it. On a 200-megahertz Pentium Pro running Linux and `gcc` 2.7.1, the THE protocol is only about 5% faster than the locking protocol. This machine's memory model requires that a memory fence instruction be inserted between lines 5 and 6 of the `pop` pseudocode. On this processor, the THE protocol spends about half of its time in the memory fence. On a 167-megahertz UltraSPARC I, however, the THE protocol is about 25% faster than the simple locking protocol. In this case we tried to quantify the performance impact of the memory fence (`membar`) instruction, too, but in all our experiments the execution times of the code with and without `membar` are about the same.

In addition to this performance advantage, because it replaces locks with memory synchronization, the THE protocol is more "nonblocking" than a straightforward locking protocol. Consequently, the THE protocol is less prone to problems that arise when spin locks are used extensively. For example, even if a worker is suspended by the operating system during the execution of `pop`, the infrequency of locking in the THE protocol means that a thief can usually complete a steal operation on the worker's deque. Recent work by Arora et al. [14] has shown that a completely nonblocking work-stealing scheduler can be implemented. Using these ideas, Lisiecki and Medina [101] have

Program	*Size*	T_1	T_∞	$\overline{P}$	c_1	T_8	T_1/T_8	T_S/T_8
fib	35	12.77	0.0005	25540	3.63	1.60	8.0	2.2
blockedmul	1024	29.9	0.0044	6730	1.05	4.3	7.0	6.6
notempmul	1024	29.7	0.015	1970	1.05	3.9	7.6	7.2
strassen	1024	20.2	0.58	35	1.01	3.54	5.7	5.6
*cilksort	$4,100,000$	5.4	0.0049	1108	1.21	0.90	6.0	5.0
†queens	22	150.	0.0015	96898	0.99	18.8	8.0	8.0
†knapsack	30	75.8	0.0014	54143	1.03	9.5	8.0	7.7
lu	2048	155.8	0.42	370	1.02	20.3	7.7	7.5
*cholesky	BCSSTK32	1427.	3.4	420	1.25	208.	6.9	5.5
heat	4096×512	62.3	0.16	384	1.08	9.4	6.6	6.1
fft	2^{20}	4.3	0.0020	2145	0.93	0.77	5.6	6.0
barnes-hut	2^{16}	124.	0.15	853	1.02	16.5	7.5	7.4

Figure 2-8: The performance of example Cilk programs. Times are in seconds and are accurate to within about 10%. The serial programs are C elisions of the Cilk programs, except for those programs that are starred (*), where the parallel program implements a different algorithm than the serial program. Programs labeled by a dagger (†) are nondeterministic, and thus, the running time on one processor is not the same as the work performed by the computation. For these programs, the value for T_1 indicates the actual work of the computation on 8 processors, and not the running time on one processor.

modified the Cilk-5 scheduler to make it completely nonblocking. Their experience is that the THE protocol greatly simplifies a nonblocking implementation.

The simplified THE protocol can be extended to support the signaling of exceptions to a worker. In Figure 2-6, the index `H` plays two roles: it marks the head of the deque, and it marks the point that the worker cannot cross when it pops. These places in the deque need not be the same. In the full THE protocol, we separate the two functions of `H` into two variables: `H`, which now only marks the head of the deque, and `E`, which marks the point that the victim cannot cross. Whenever $\mathtt{E} > \mathtt{T}$, some exceptional condition has occurred, which includes the frame being stolen, but it can also be used for other exceptions. For example, setting $\mathtt{E} = \infty$ causes the worker to discover the exception at its next pop. In the new protocol, `E` replaces `H` in line 6 of the worker/victim. Moreover, lines 7–15 of the worker/victim are replaced by a call to an ***exception handler*** to determine the type of exception (stolen frame or otherwise) and the proper action to perform. The thief code is also modified. Before trying to steal, the thief increments `E`. If there is nothing to steal, the thief restores `E` to the original value. Otherwise, the thief steals frame `H` and increments `H`. From the point of view of a worker, the common case is the same as in the simplified protocol: it compares two pointers (`E` and `T` rather than `H` and `T`).

The exception mechanism is used to implement `abort`. When a Cilk procedure executes an `abort` instruction, the runtime system serially walks the tree of outstanding descendants of that procedure. It marks the descendants as aborted and signals an abort exception on any processor working on a descendant. At its next `pop`, an aborted procedure will discover the exception, notice that it has been aborted, and return immediately. It is conceivable that a procedure could run for a long time without executing a pop and discovering that it has been aborted. We made the design decision to accept the possibility of this unlikely scenario, figuring that more cycles were likely to be lost in work overhead if we abandoned the THE protocol for a mechanism that solves this minor problem.

2.7 Benchmarks

In this section, we evaluate the performance of Cilk-5. We show that on 12 applications, the work overhead c_1 is close to 1, which indicates that the Cilk-5 implementation exploits the work-first principle effectively and achieves the goal of "scaling down" to 1 processor. We then present a breakdown of Cilk's work overhead c_1 on four machines. Finally, we present experiments showing that Cilk applications "scale up" as well, and that the critical-path overhead c_∞ is reasonably small. Our experiments show that Cilk delivers both high performance and portability, at least on the SMP machines targeted by the Cilk-5 implementation.

Figure 2-8 shows a table of performance measurements taken for 12 Cilk programs on a Sun Enterprise 5000 SMP with 8 167-megahertz UltraSPARC processors, each with 512 kilobytes of L2 cache, 16 kilobytes each of L1 data and instruction caches, running Solaris 2.5. We compiled our programs with gcc 2.7.2 at optimization level -O3. For a full description of these programs, see the Cilk 5.1 manual [38]. The table shows the work of each Cilk program T_1, the critical path T_∞, and the two derived quantities $\overline{P}$ and c_1. The table also lists the running time T_8 on 8 processors, and the speedup T_1/T_8 relative to the one-processor execution time, and speedup T_S/T_8 relative to the serial execution time.

For the 12 programs, the parallelism $\overline{P}$ is in most cases quite large relative to the number of processors on a typical SMP. These measurements validate our assumption of parallel slackness, which implies that the work term dominates in Inequality (2.4). For instance, on 1024×1024 matrices, notempmul runs with a parallelism of 1970—yielding adequate parallel slackness for up to several hundred processors. For even larger machines, one normally would not run such a small problem. For notempmul, as well as the other 11 applications, the parallelism grows with problem size, and thus sufficient parallel slackness is likely to exist even for much larger machines, as long as the problem sizes are scaled appropriately.

The work overhead c_1 is only a few percent larger than 1 for most programs, which shows that, by faithfully implementing the work-first principle, Cilk-5 does not introduce significant overheads when sequential programs are parallelized. The two cases where the work overhead is larger (cilksort and cholesky) are due to the fact that we had to change the serial algorithm to obtain a parallel algorithm, and thus the comparison is not against the C elision. For example, the serial C algorithm for sorting is an in-place quicksort, but the parallel algorithm cilksort requires an additional temporary array which adds overhead beyond the overhead of Cilk itself. Similarly, our

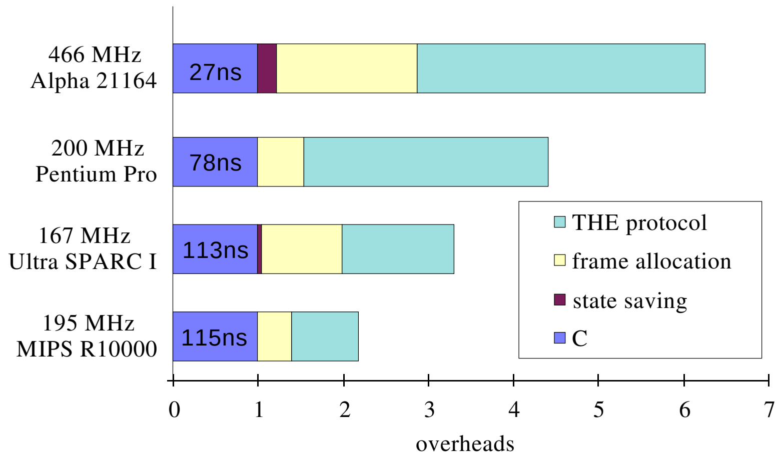


Figure 2-9: Breakdown of overheads for `fib` running on one processor on various architectures. The overheads are normalized to the running time of the serial C elision. The three overheads are for saving the state of a procedure before a spawn, the allocation of activation frames for procedures, and the THE protocol. Absolute times are given for the per-spawn running time of the C elision.

parallel Cholesky factorization uses a quadtree representation of the sparse matrix, which induces more work than the linked-list representation used in the serial C algorithm. Finally, the work overhead for `fib` is large, because `fib` does essentially no work besides spawning procedures. Thus, the overhead $c_1 = 3.63$ for `fib` gives a good estimate of the cost of a Cilk `spawn` versus a traditional C function call. With such a small overhead for spawning, one can understand why for most of the other applications, which perform significant work for each spawn, the overhead of Cilk-5's scheduling is barely noticeable compared to the 10% "noise" in our measurements.

We now present a breakdown of Cilk's serial overhead c_1 into its components. Because scheduling overheads are small for most programs, we perform our analysis with the `fib` program from Figure 2-1. This program is unusually sensitive to scheduling overheads, because it contains little actual computation. We give a breakdown of the serial overhead into three components: the overhead of saving state before spawning, the overhead of allocating activation frames, and the overhead of the THE protocol.

Figure 2-9 shows the breakdown of Cilk's serial overhead for `fib` on four machines. Our methodology for obtaining these numbers is as follows. First, we take the serial C `fib` program and time its execution. Then, we individually add in the code that generates each of the overheads and time the execution of the resulting program. We attribute the additional time required by the modified program to the scheduling code we added. In order to verify our numbers, we timed the `fib` code with all of the Cilk overheads added (the code shown in Figure 2-5), and compared the resulting time to the sum of the individual overheads. In all cases, the two times differed by less than 10%.

Overheads vary across architectures, but the overhead of Cilk is typically only a few times the C running time on this spawn-intensive program. Overheads on the Alpha machine are particularly

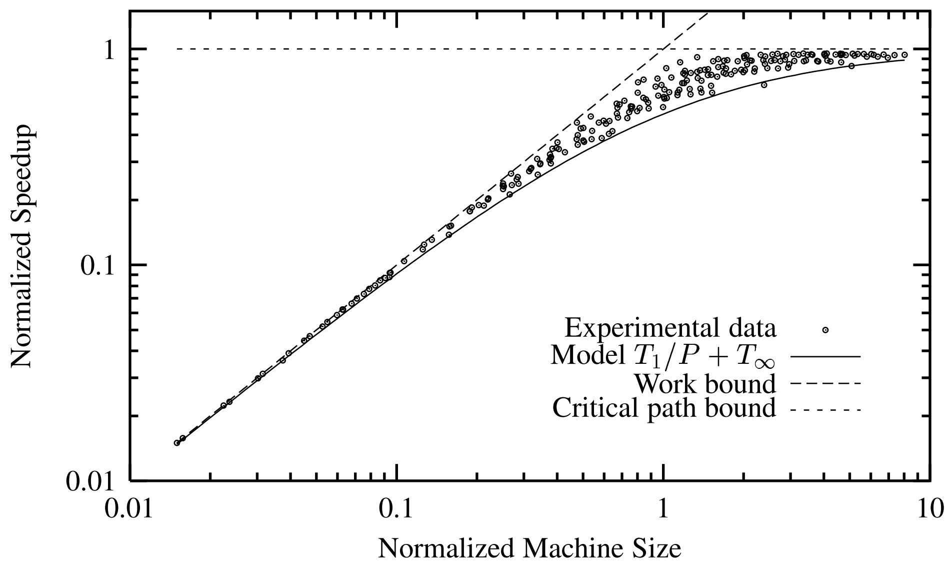


Figure 2-10: Normalized speedup curve for Cilk-5. The horizontal axis is the number P of processors and the vertical axis is the speedup T_1/T_P, but each data point has been normalized by dividing by T_1/T_∞. The graph also shows the speedup predicted by the formula $T_P = T_1/P + T_\infty$.

large, because its native C function calls are fast compared to the other architectures. The state-saving costs are small for `fib`, because all four architectures have write buffers that can hide the latency of the writes required.

We also attempted to measure the critical-path overhead c_∞. We used the synthetic `knary` benchmark [29] to synthesize computations artificially with a wide range of work and critical-path lengths. Figure 2-10 shows the outcome from many such experiments. The figure plots the measured speedup T_1/T_P for each run against the machine size P for that run. In order to plot different computations on the same graph, we normalized the machine size and the speedup by dividing these values by the parallelism $\overline{P} = T_1/T_\infty$, as was done in [29]. For each run, the horizontal position of the plotted datum is the inverse of the slackness $P/\overline{P}$, and thus, the normalized machine size is 1.0 when the number of processors is equal to the parallelism. The vertical position of the plotted datum is $(T_1/T_P)/\overline{P} = T_\infty/T_P$, which measures the fraction of maximum obtainable speedup. As can be seen in the chart, for almost all runs of this benchmark, we observed $T_P \leq T_1/P + 1.0T_\infty$. (One exceptional data point satisfies $T_P \approx T_1/P + 1.05T_\infty$.) Thus, although the work-first principle caused us to move overheads to the critical path, the ability of Cilk applications to scale up was not significantly compromised.

2.8 Related work

Mohr *et al.* [112] introduced lazy task creation in their implementation of the Mul-T language. Lazy task creation is similar in many ways to our lazy scheduling techniques. Mohr *et al.* report a work overhead of around 2 when comparing with serial T, the Scheme dialect on which Mul-T is based. Our research confirms the intuition behind their methods and shows that work overheads of close to 1 are achievable.

The Cid language [118] is like Cilk in that it uses C as a base language and has a simple preprocessing compiler to convert parallel Cid constructs to C. Cid is designed to work in a distributed memory environment, and so it employs latency-hiding mechanisms which Cilk-5 could avoid. Both Cilk and Cid recognize the attractiveness of basing a parallel language on C so as to leverage C compiler technology for high-performance codes. Cilk is a faithful extension of C, however, supporting the simplifying notion of a C elision and allowing Cilk to exploit the C compiler technology more readily.

TAM [45] and Lazy Threads [68] also analyze many of the same overhead issues in a more general, "nonstrict" language setting, where the individual performances of a whole host of mechanisms are required for applications to obtain good overall performance. In contrast, Cilk's multithreaded language provides an execution model based on work and critical-path length that allows us to focus our implementation efforts by using the work-first principle. Using this principle as a guide, we have concentrated our optimizing effort on the common-case protocol code to develop an efficient and portable implementation of the Cilk language.

2.9 Conclusion

Cilk is the superior programming tool of choice for discriminating hackers.

(Directors of the ICFP'98 Programming Contest)

The Cilk system that we discussed in this chapter effectively attains portable high-performance of parallel programs. Cilk achieves high performance because of a provably efficient parallel scheduler and an implementation aimed at the systematic reduction of common-case overheads. Rather than determining the common case experimentally, we derived the work-first principle, which guides the optimization effort of the system.

Cilk attains portability because of a clean language and an algorithmic performance model that predicts the execution time of a program in terms of work and critical-path length. Both these measures can be analyzed with well-known techniques from conventional algorithmic analysis, and the critical-path length is really not more difficult to analyze than the work. In this way, we can design algorithms for portability by choosing an algorithm with the most appropriate work and/or critical path.

The simplicity of the Cilk language contributes to portability because a C user does not need to learn too many linguistic constructs in order to write a parallel program. Like users of high-level languages such as Multilisp [75], Mul-T [94], Id [119], pH [117], NESL [23], ZPL [34], and High Performance Fortran [93, 80], a Cilk user is not expected to write protocols. With message-passing systems such as MPI [134] and PVM [62], on the contrary, a programmer must write protocols and worry about deadlocks and buffer overflows. Cilk is a "simple" language. Although simplicity is hard to quantify, a simple language such as Cilk reduces the "barriers to entry" to parallelism and opens an evolutionary path to a world where most programs can be run indifferently on parallel and sequential machines.

Chapter 3

Cache-oblivious algorithms

With Cilk, as discussed in Chapter 2, we can design "processor-oblivious" algorithms and write programs that run efficiently on any number of processors in the range of interest. Cilk tackles the problem of portable high performance from the point of view of how to cope with parallelism. In this chapter, we focus on a complementary aspect of portable high performance, namely, how to deal portably with the memory hierarchy. In this chapter we forget about parallelism, and we deal with sequential algorithms only. We shall attempt a grand unification of these two topics in Chapter 4.

This chapter is about optimal ***cache-oblivious*** algorithms, in which no variables dependent on hardware parameters, such as cache size and cache-line length, need to be tuned to achieve optimality. In this way, these algorithms are by design efficient and portable across different implementations of the memory hierarchy. We study asymptotically optimal cache-oblivious algorithms for rectangular matrix transpose and multiplication, FFT, and sorting on computers with multiple levels of caching. For a cache with size Z and cache-line length L where $Z = \Omega(L^2)$ the number of cache misses for an $m \times n$ matrix transpose is $\Theta(1 + mn/L)$. The number of cache misses for either an n-point FFT or the sorting of n numbers is $\Theta(1 + (n/L)(1 + \log_Z n))$. A straightforward generalization of the `matrixmul` algorithm from Section 2.4 yields an $\Theta(mnp)$-work algorithm to multiply an $m \times n$ matrix by an $n \times p$ matrix that incurs $\Theta(1 + (mn + np + mp)/L + mnp/L\sqrt{Z})$ cache faults.

The cache-oblivious algorithms we study are all divide-and-conquer. In Cilk, divide and conquer is useful because it generates parallelism recursively so that the critical path of divide-and-conquer algorithms is typically some polylogarithmic function of the work. For cache-oblivious algorithms, divide-and-conquer plays the complementary role of splitting the original problem into smaller problems that eventually fit into cache. Once the problem is small enough, it can be solved

This chapter represents joint work with Charles Leiserson, Harald Prokop, and Sridhar Ramachandran. A preliminary version appears in [56].

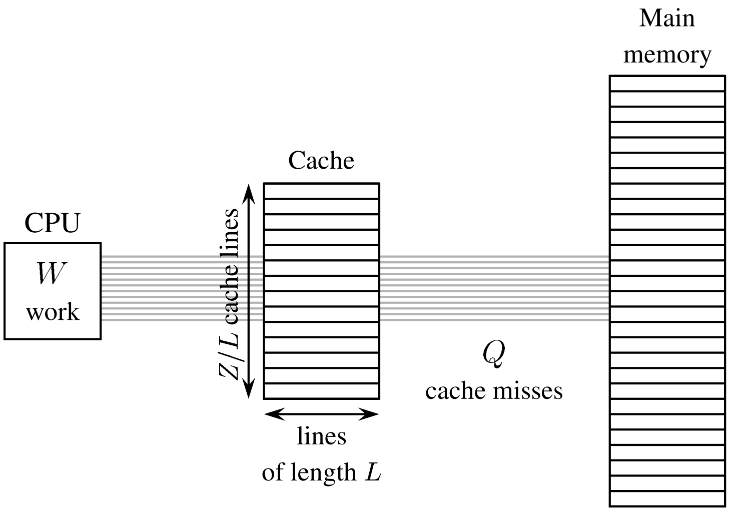


Figure 3-1: The ideal-cache model

with the optimal number of cache misses—those required to read the input and write the output. Because of these two effects, divide and conquer is a powerful design technique for portable high-performance programs.

This chapter is entirely theoretical, and it lays down a foundation for understanding cache-oblivious algorithms. As it is customary in theoretical investigations in computer science, we will focus on asymptotic analysis and disregard constant factors. While imperfect, this kind of analysis offers insights on the principles underlying cache-oblivious algorithms, so that we can apply similar ideas to other problems. We will apply this theory of cache-oblivious algorithms in Chapter 6 in the context of FFTW's "register-oblivious" scheduler of Fourier transform algorithms.

Before discussing the notion of cache obliviousness more precisely, we first introduce the $\boldsymbol{(Z, L)}$ ***ideal-cache model*** to study the cache complexity of algorithms. This model, which is illustrated in Figure 3-1, consists of a computer with a two-level memory hierarchy consisting of an ideal (data) cache of Z words and an arbitrarily large main memory. Because the actual size of words in a computer is typically a small, fixed size (4 bytes, 8 bytes, etc.), we shall assume that the word size is constant; the particular constant does not affect our asymptotic analyses. The cache is partitioned into ***cache lines***, each consisting of L consecutive words that are always moved together between cache and main memory. Cache designers typically use $L > 1$, banking on spatial locality to amortize the overhead of moving the cache line. We shall generally assume that the cache is ***tall***:

$$Z = \Omega(L^2) , \tag{3.1}$$

which is usually true in practice.

The processor can only reference words that reside in the cache. If the referenced word belongs to a line already in cache, a ***cache hit*** occurs, and the word is delivered to the processor. Otherwise, a ***cache miss*** occurs, and the line is fetched into the cache. The ideal cache is ***fully associative*** [79, Ch. 5]: cache lines can be stored anywhere in the cache. If the cache is full, a cache line must be evicted. The ideal cache uses the optimal off-line strategy of replacing the cache line whose next access is farthest in the future [18], and thus it exploits temporal locality perfectly.

An algorithm with an input of size n is measured in the ideal-cache model in terms of its ***work complexity*** $W(n)$—its conventional running time in a RAM model [8]—and its ***cache complexity*** $Q(n; Z, L)$—the number of cache misses it incurs as a function of the size Z and line length L of the ideal cache. When Z and L are clear from context, we denote the cache complexity as simply $Q(n)$ to ease notation. The "work" W measure in this chapter is the same as the "work" T_1 measure from Chapter 2; we are switching notation because in this chapter we have no notion of parallelism that justifies the notation T_1.

The ideal-cache model glosses over the fact that most real caches are not fully associative, they do not employ the optimal replacement strategy, and they are sometimes write-through.[1] Nevertheless, this model is a good approximation to many real systems. For example, the register set of a processor can be seen as a fully associative cache controlled by an omniscient compiler. In the same way, an operating system that swaps memory pages to disk can amortize the overheads of full associativity against the expensive I/O, and the optimal replacement strategy can be simulated using a least-recently-used (LRU) policy.[2] (See [133] and Section 3.5.) Furthermore, if an algorithm does not run well with an ideal cache, it won't run well with a less-than-ideal cache either, and thus the model can be helpful to prove lower bounds. In this chapter, however, we are interested in proving upper bound results on the cache complexity, and we assume that the ideal-cache assumptions hold.

We define an algorithm to be ***cache aware*** if it contains parameters (set at either compile-time or runtime) that can be tuned to optimize the cache complexity for the particular cache size and line length. Otherwise, the algorithm is ***cache oblivious***. Historically, good performance has been obtained using cache-aware algorithms, but we shall exhibit several cache-oblivious algorithms that are asymptotically as efficient as their cache-aware counterparts.

To illustrate the notion of cache awareness, consider the problem of multiplying two $n \times n$ matrices A and B to produce their $n \times n$ product C. We assume that the three matrices are stored in row-major order, as shown in Figure 3-2(a). We further assume that n is "big," i.e. $n > L$ in order to simplify the analysis. The conventional way to multiply matrices on a computer with caches is to use a ***blocked*** algorithm [69, p. 45]. The idea is to view each matrix M as consisting of $(n/s) \times (n/s)$ submatrices M_{ij} (the blocks), each of which has size $s \times s$, where s is a tuning parameter. The following algorithm implements this strategy:

---

[1]A ***write-through*** cache transmits writes to the next level of the memory hierarchy immediately [79].

[2]Page replacement in current operating systems is constrained by the low associativity of the L2 cache, however. If the ***page coloring*** technique [106] is used, the operating system improves the behavior of the L2 cache, but it cannot implement the LRU policy exactly.

```
BLOCK-MULT(A, B, C, n)
for i ← 1 to n/s
    do for j ← 1 to n/s
        do for k ← 1 to n/s
            do ORD-MULT(A_ik, B_kj, C_ij, s)
```

where ORD-MULT(A, B, C, s) is a subroutine that computes $C \leftarrow C + AB$ on $s \times s$ matrices using the ordinary $O(s^3)$ algorithm. (This algorithm assumes for simplicity that s evenly divides n, but in practice s and n need have no special relationship, which yields more complicated code in the same spirit.)

Depending on the cache size of the machine on which BLOCK-MULT is run, the parameter s can be tuned to make the algorithm run fast, and thus BLOCK-MULT is a cache-aware algorithm. To minimize the cache complexity, we choose s so that the three $s \times s$ submatrices simultaneously fit in cache. An $s \times s$ submatrix is stored on $\Theta(s + s^2/L)$ cache lines. From the tall-cache assumption (3.1), we can see that $s = \Theta(\sqrt{Z})$. Thus, each of the calls to ORD-MULT runs with at most $Z/L = \Theta(s^2/L)$ cache misses needed to bring the three matrices into the cache. Consequently, the cache complexity of the entire algorithm is $\Theta(1 + n^2/L + (n/\sqrt{Z})^3(Z/L)) = \Theta(1 + n^2/L + n^3/L\sqrt{Z})$, since the algorithm has to read n^2 elements, which reside on $\lceil n^2/L \rceil$ cache lines.

The same bound can be achieved using a simple cache-oblivious algorithm that requires no tuning parameters such as the s in BLOCK-MULT. We present such an algorithm, which works on general rectangular matrices, in Section 3.1. The problems of computing a matrix transpose and of performing an FFT also succumb to remarkably simple algorithms, which are described in Section 3.2. Cache-oblivious sorting poses a more formidable challenge. In Sections 3.3 and 3.4, we present two sorting algorithms, one based on mergesort and the other on distribution sort, both which are optimal.

The ideal-cache model makes the perhaps questionable assumption that memory is managed automatically by an *optimal* cache replacement strategy. Although the current trend in architecture does favor automatic caching over programmer-specified data movement, Section 3.5 addresses this concern theoretically. We show that the assumptions of another hierarchical memory model in the literature, in which memory movement is programmed explicitly, are actually no weaker than ours. Specifically, we prove (with only minor assumptions) that optimal cache-oblivious algorithms in the ideal-cache model are also optimal in the serial uniform memory hierarchy (SUMH) model [11, 148]. Section 3.6 discusses related work, and Section 3.7 offers some concluding remarks.

3.1 Matrix multiplication

This section describes an algorithm for multiplying an $m \times n$ by an $n \times p$ matrix cache-obliviously using $\Theta(mnp)$ work and incurring $\Theta(1 + (mn + np + mp)/L + mnp/L\sqrt{Z})$ cache misses. These results require the tall-cache assumption (3.1) for matrices stored with in a row-major layout format, but the assumption can be relaxed for certain other layouts. We also discuss Strassen's algorithm [138] for multiplying $n \times n$ matrices, which uses $\Theta(n^{\lg 7})$ work[3] and incurs $\Theta(1 + n^2/L + n^{\lg 7}/L\sqrt{Z})$ cache misses.

To multiply a $m \times n$ matrix A and a $n \times p$ matrix B, the algorithm halves the largest of the three dimensions and recurs according to one of the following three cases:

$$\begin{aligned}
&\textbf{(a)} \quad AB = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix} , \\
&\textbf{(b)} \quad AB = \begin{pmatrix} A_1 & A_2 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = A_1 B_1 + A_2 B_2 , \\
&\textbf{(c)} \quad AB = A \begin{pmatrix} B_1 & B_2 \end{pmatrix} = \begin{pmatrix} AB_1 & AB_2 \end{pmatrix} .
\end{aligned}$$

In case (a), we have $m \geq \max\{n, p\}$. Matrix A is split horizontally, and both halves are multiplied by matrix B. In case (b), we have $n \geq \max\{m, p\}$. Both matrices are split, and the two halves are multiplied. In case (c), we have $p \geq \max\{m, n\}$. Matrix B is split vertically, and each half is multiplied by A. For square matrices, these three cases together are equivalent to the recursive multiplication algorithm described in [26]. The base case occurs when $m = n = p = 1$, in which case the two elements are multiplied and added into the result matrix.

It can be shown by induction that the work of this algorithm is $O(mnp)$, the same as the standard matrix multiplication algorithm. Although this straightforward divide-and-conquer algorithm contains no tuning parameters, it uses cache optimally. To analyze the cache complexity of the algorithm, we assume that the three matrices are stored in row-major order, as shown in Figure 3-2(a). We further assume that any row in each of the matrices does not fit in 1 cache line, that is, $\min\{m, n, p\} \geq L$. (We omit the analysis of the general case because it does not offer any new insight. See [125] for the complete proof.)

The following recurrence describes the cache complexity:

$$Q(m, n, p) \leq \begin{cases} O((mn + np + mp)/L) & \text{if } (mn + np + mp) \leq \alpha Z , \\ 2Q(m/2, n, p) + O(1) & \text{if } m \geq n \text{ and } m \geq p , \\ 2Q(m, n/2, p) + O(1) & \text{if } n > m \text{ and } n \geq p , \\ 2Q(m, n, p/2) + O(1) & \text{otherwise} , \end{cases} \tag{3.2}$$

[3]We use the notation lg to denote $\log_2$.

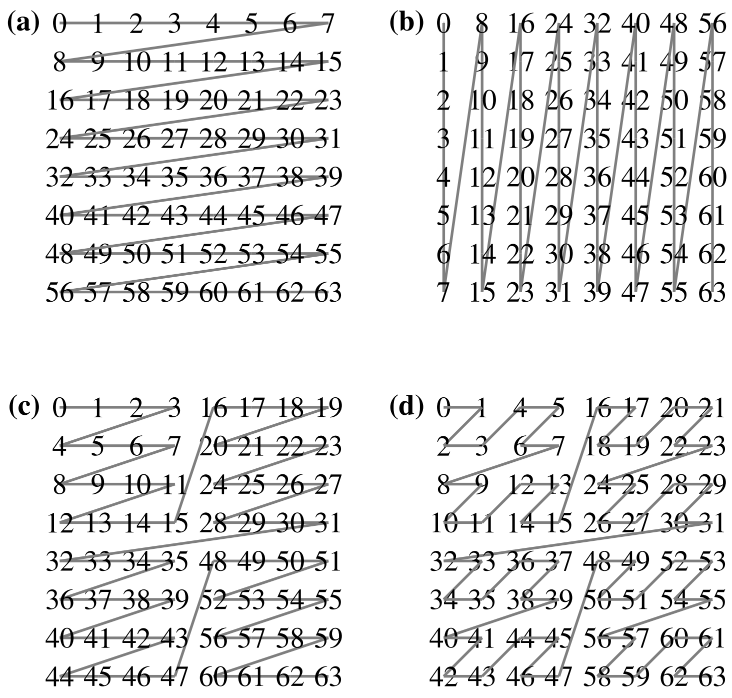


Figure 3-2: Layout of a 16×16 matrix in **(a)** row major, **(b)** column major, **(c)** 4×4-blocked, and **(d)** bit-interleaved layouts.

where α is a constant chosen sufficiently small to allow the three submatrices (and whatever small number of temporary variables there may be) to fit in the cache. The base case arises as soon as all three matrices fit in cache. Using reasoning similar to that for analyzing ORD-MULT within BLOCK-MULT, the matrices are held on $\Theta((mn+np+mp)/L)$ cache lines, assuming a tall cache. Thus, the only cache misses that occur during the remainder of the recursion are the $\Theta((mn + np + mp)/L)$ cache misses that occur when the matrices are brought into the cache. The recursive case arises when the matrices do not fit in cache, in which case we pay for the cache misses of the recursive calls, which depend on the dimensions of the matrices, plus $O(1)$ cache misses for the overhead of manipulating submatrices. The solution to this recurrence is $Q(m, n, p) = O(1 + (mn + np + mp)/L + mnp/L\sqrt{Z})$, which is the same as the cache complexity of the cache-aware BLOCK-MULT algorithm for square matrices. Intuitively, the cache-oblivious divide-and-conquer algorithm uses cache effectively because once a subproblem fits into the cache, no more cache misses occur for smaller subproblems.

We require the tall-cache assumption (3.1) in this analysis because the matrices are stored in row-major order. Tall caches are also needed if matrices are stored in column-major order (Figure 3-2(b)), but the assumption that $Z = \Omega(L^2)$ can be relaxed for certain other matrix layouts. The $s \times s$-blocked layout (Figure 3-2(c)), for some tuning parameter s, can be used to achieve the same bounds with the weaker assumption that the cache holds at least some sufficiently large constant number of lines. The cache-oblivious bit-interleaved layout (Figure 3-2(d)) has the same advantage as the blocked layout, but no tuning parameter need be set, since submatrices of size $\Theta(\sqrt{L} \times \sqrt{L})$ are cache-obliviously stored on one cache line. The advantages of bit-interleaved and related layouts have been studied in [53] and [35, 36]. One of the practical disadvantages of bit-interleaved layouts is that index calculations on today's conventional microprocessors can be costly.

For square matrices, the cache complexity $Q(n) = \Theta(1 + n^2/L + n^3/L\sqrt{Z})$ of the cache-oblivious matrix multiplication algorithm matches the lower bound by Hong and Kung [82]. This lower bound holds for all algorithms that execute the $\Theta(n^3)$ operations given by the definition of matrix multiplication

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} .$$

No tight lower bounds for the general problem of matrix multiplication are known. By using an asymptotically faster algorithm, such as Strassen's algorithm [138] or one of its variants [152], both the work and cache complexity can be reduced. Indeed, Strassen's algorithm, which is cache oblivious, can be shown to have cache complexity $O(1 + n^2/L + n^{\lg 7}/L\sqrt{Z})$.

3.2 Matrix transposition and FFT

This section describes a cache-oblivious algorithm for transposing a $m \times n$ matrix that uses $O(mn)$ work and incurs $O(1 + mn/L)$ cache misses, which is optimal. Using matrix transposition as a subroutine, we convert a variant [150] of the "six-step" fast Fourier transform (FFT) algorithm [17] into an optimal cache-oblivious algorithm. This FFT algorithm uses $O(n \lg n)$ work and incurs $O\left(1 + (n/L)\left(1 + \log_Z n\right)\right)$ cache misses.

The problem of matrix transposition is defined as follows. Given an $m \times n$ matrix stored in a row-major layout, compute and store A^T into an $n \times m$ matrix B also stored in a row-major layout. The straightforward algorithm for transposition that employs doubly nested loops incurs $\Theta(mn)$ cache misses on one of the matrices when $mn \gg Z$, which is suboptimal.

Optimal work and cache complexities can be obtained with a divide-and-conquer strategy, however. If $n \geq m$, we partition

$$A = (A_1 \; A_2) \; , \quad B = \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} .$$

Then, we recursively execute $\textsc{Transpose}(A_1, B_1)$ and $\textsc{Transpose}(A_2, B_2)$. If $m > n$, we divide matrix A horizontally and matrix B vertically and likewise perform two transpositions recursively.

The next two lemmas provide upper and lower bounds on the performance of this algorithm.

Lemma 1 *The cache-oblivious matrix-transpose algorithm uses $O(mn)$ work and incurs $O(1 + mn/L)$ cache misses for an $m \times n$ matrix.*

Proof: We omit the proof that the algorithm uses $O(mn)$ work. For the cache analysis, let $Q(m, n)$ be the cache complexity of transposing a $m \times n$ matrix. We assume that the matrices are stored in row-major order, the column-major case having a similar analysis.

Let α be a constant sufficiently small such that two submatrices of size $m \times n$ and $n \times m$, where $\max\{m, n\} \leq \alpha L$, fit completely in the cache even if each row is stored in a different cache line. Such a constant exists because of the tall-cache assumption. We distinguish the following three cases.

Case I: $\max\{m, n\} \leq \alpha L$.

Both matrices fit in $O(1) + 2mn/L$ lines. If α is small enough, two matrices fit completely in cache, and we only need to read and/or write each line once in order to complete the transposition. Therefore $Q(m, n) = O(1 + mn/L)$.

Case II: $m \leq \alpha L < n$ OR $n \leq \alpha L < m$.

For this case, assume first that $m \leq \alpha L < n$. The transposition algorithm divides the greater dimension n by 2 and performs divide and conquer. At some point in the recursion, n is in the range $\alpha L/2 \leq n \leq \alpha L$, and the whole problem fits in cache as in Case I. Because the layout is row-major, at this point the input array has n rows, m columns, and it is laid out in contiguous locations, thus requiring at most $O(1 + nm/L)$ cache misses to be read. The output array consists of nm elements in m rows, where in the worst case every row lies on a different cache line. Consequently, we incur at most $O(m + nm/L)$ misses for writing the output array. Since $\alpha L \geq n \geq \alpha L/2$, the total cache complexity for this base case is $O(1 + m)$.

These observations yield the recurrence

$$Q(m, n) \leq \begin{cases} O(1 + m) & \text{if } n \in [\alpha L/2, \alpha L] \text{ ,} \\ 2Q(m, n/2) + O(1) & \text{otherwise ,} \end{cases}$$

whose solution is $Q(m, n) = O(1 + mn/L)$.

The case $n \leq \alpha L < m$ is analogous.

Case III: $m, n > \alpha L$.

As in Case II, at some point in the recursion both n and m are in the range $[\alpha L/2, \alpha L]$. The whole problem fits into cache and it can be solved with at most $O(m + n + mn/L)$ cache misses.

The cache complexity thus satisfies the recurrence

$$Q(m,n) \le \begin{cases} O(m+n+mn/L) & \text{if } m,n \in [\alpha L/2, \alpha L] \,, \\ 2Q(m/2,n)+O(1) & \text{if } m \ge n \,, \\ 2Q(m,n/2)+O(1) & \text{otherwise,} \end{cases}$$

whose solution is $Q(m,n) = O(1+mn/L)$.

■

Theorem 2 *The cache-oblivious matrix-transpose algorithm is asymptotically optimal.*

Proof: For an $m \times n$ matrix, the matrix-transposition algorithm must write to mn distinct elements, which occupy at least $\lceil mn/L \rceil = \Omega(1+mn/L)$ cache lines. ■

As an example of application of the cache-oblivious transposition algorithm, in the rest of this section we describe and analyze a cache-oblivious algorithm for computing the discrete Fourier transform of a complex array of n elements, where n is an exact power of 2. The basic algorithm is the well-known "six-step" variant [17, 150] of the Cooley-Tukey FFT algorithm [41]. Using the cache-oblivious transposition algorithm, however, the FFT becomes cache-oblivious, and its performance matches the lower bound by Hong and Kung [82].

Recall that the ***discrete Fourier transform (DFT)*** of an array X of n complex numbers is the array Y given by

$$Y[i] = \sum_{j=0}^{n-1} X[j]\omega_n^{-ij} \,, \tag{3.3}$$

where $\omega_n = e^{2\pi\sqrt{-1}/n}$ is a primitive nth root of unity, and $0 \le i < n$.

Many known algorithms evaluate Equation (3.3) in time $O(n \lg n)$ for all integers n [48]. In this section, however, we assume that n is an exact power of 2, and compute Equation (3.3) according to the Cooley-Tukey algorithm, which works recursively as follows. In the base case where $n = O(1)$, we compute Equation (3.3) directly. Otherwise, for any factorization $n = n_1 n_2$ of n, we have

$$Y[i_1 + i_2 n_1] = \sum_{j_2=0}^{n_2-1} \left[\left(\sum_{j_1=0}^{n_1-1} X[j_1 n_2 + j_2]\omega_{n_1}^{-i_1 j_1} \right) \omega_n^{-i_1 j_2} \right] \omega_{n_2}^{-i_2 j_2} \,. \tag{3.4}$$

Observe that both the inner and the outer summation in Equation (3.4) is a DFT. Operationally, the computation specified by Equation (3.4) can be performed by computing n_2 transforms of size n_1

(the inner sum), multiplying the result by the factors $\omega_n^{-i_1 j_2}$ (called the ***twiddle factors*** [48]), and finally computing n_1 transforms of size n_2 (the outer sum).

We choose n_1 to be $2^{\lceil \lg n/2 \rceil}$ and n_2 to be $2^{\lfloor \lg n/2 \rfloor}$. The recursive step then operates as follows.

1. Pretend that input is a row-major $n_1 \times n_2$ matrix A. Transpose A in place, i.e., use the cache-oblivious algorithm to transpose A onto an auxiliary array B, and copy B back onto A. Notice that if $n_1 = 2n_2$, we can consider the matrix to be made up of records containing two elements.

2. At this stage, the inner sum corresponds to a DFT of the n_2 rows of the transposed matrix. Compute these n_2 DFT's of size n_1 recursively. Observe that, because of the previous transposition, we are transforming a contiguous array of elements.

3. Multiply A by the twiddle factors, which can be computed on the fly with no extra cache misses.

4. Transpose A in place, so that the inputs to the next stage is arranged in contiguous locations.

5. Compute n_1 DFT's of the rows of the matrix, recursively.

6. Transpose A in place, so as to produce the correct output order.

It can be proven by induction that the work complexity of this FFT algorithm is $O(n \lg n)$. We now analyze its cache complexity. The algorithm always operates on contiguous data, by construction. In order to simplify the analysis of the cache complexity, assume a tall cache, in which case each transposition operation and the multiplication by the twiddle factors require at most $O(1+n/L)$ cache misses. Thus, the cache complexity satisfies the recurrence

$$Q(n) \leq \begin{cases} O(1+n/L), & \text{if } n \leq \alpha Z \ , \\ n_1 Q(n_2) + n_2 Q(n_1) + O(1+n/L) & \text{otherwise} \ , \end{cases} \tag{3.5}$$

for a sufficiently small constant α chosen such that a subproblem of size αZ fits in cache. This recurrence has solution

$$Q(n) = O\left(1 + (n/L)\left(1 + \log_Z n\right)\right) \ ,$$

which is asymptotically optimal for a Cooley-Tukey algorithm, matching the lower bound by Hong and Kung [82] when n is an exact power of 2. As with matrix multiplication, no tight lower bounds for cache complexity are known for the general problem of computing the DFT.

This cache-oblivious FFT algorithm will be used in FFTW in Chapter 6. Even if the ideal-cache model is not a precise description of L1 or L2 caches, the register set of a processor is a good approximation to an ideal cache with $L = 1$. Registers constitute the "cache," the rest of the memory hierarchy constitutes the "main memory," and a compiler can usually approximate the optimal replacement policy when allocating registers because it knows the full instruction sequence. `genfft` uses this cache-oblivious FFT algorithm to produce portable C code that can be compiled with the asymptotically optimal number of register spills, independently of the size of the register set.

A "radix-2" or any other "constant-radix" FFT algorithm would not be asymptotically optimal. These algorithms reduce a problem of size n into n_1 subproblems of size n/n_1, for some constant n_1, while the optimal cache-oblivious algorithm produces a nonconstant number of subproblems. To see why a constant-radix algorithm is nonoptimal, we can solve Equation (3.5) for the case where n_1 is a constant. The resulting cache complexity $O\left(1 + (n/L)\left(1 + \lg(n/Z)\right)\right)$ is asymptotically suboptimal.

3.3 Funnelsort

Although cache oblivious, algorithms like the familiar two-way merge sort and the Cilksort variant from Section 2.4 are not asymptotically optimal with respect to cache misses. Like the constant-radix FFT algorithm from Section 3.2, they divide a problem into a constant number of subproblems, and their resulting cache complexity is suboptimal. The Z-way mergesort mentioned by Aggarwal and Vitter [6] is optimal in terms of cache complexity, but it is cache aware. This section describes a cache-oblivious sorting algorithm called "funnelsort." This algorithm has an asymptotically optimal work complexity $O(n \lg n)$, as well as an optimal cache complexity $O\left(1 + (n/L)\left(1 + \log_Z n\right)\right)$ if the cache is tall.

Like Cilksort, funnelsort is a variant of mergesort. In order to sort a (contiguous) array of n elements, funnelsort performs the following two steps:

1. Split the input into $n^{1/3}$ contiguous arrays of size $n^{2/3}$, and sort these arrays recursively.
2. Merge the $n^{1/3}$ sorted sequences using a $n^{1/3}$-merger, which is described below.

Funnelsort differs from mergesort in the way the merge operation works. Merging is performed by a device called a ***k-merger***, which inputs k sorted sequences and merges them. A k-merger operates by recursively merging sorted sequences that become progressively longer as the algorithm proceeds. Unlike mergesort, however, a k-merger stops working on a merging subproblem when the merged output sequence becomes "long enough," and it resumes working on another merging subproblem.

Since this complicated flow of control makes a k-merger a bit tricky to describe, we explain the operation of the k-merger pictorially. Figure 3-3 shows a representation of a k-merger, which has k sorted sequences as inputs. Throughout its execution, the k-merger maintains the following invariant.

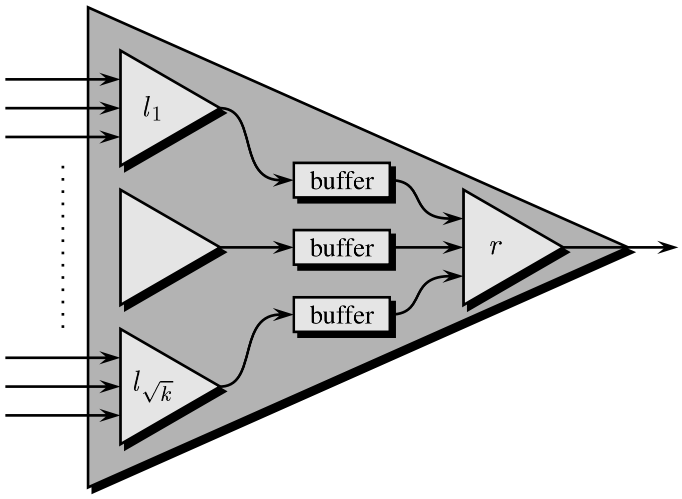


Figure 3-3: Illustration of a k-merger. A k-merger (dark in the figure) is built recursively out of $\sqrt{k}$ "left" $\sqrt{k}$-mergers $l_1, l_2, \ldots, l_{\sqrt{k}}$, a series of buffers, and one "right" $\sqrt{k}$-merger r.

Invariant *The invocation of a k-merger outputs the first k^3 elements of the sorted sequence obtained by merging the k input sequences.*

A k-merger is built recursively out of $\sqrt{k}$-mergers in the following way. The k inputs are partitioned into $\sqrt{k}$ sets of $\sqrt{k}$ elements, and these sets form the input to the $\sqrt{k}$ $\sqrt{k}$-mergers $l_1, l_2, \ldots, l_{\sqrt{k}}$ in the left part of the figure. The outputs of these mergers are connected to the inputs of $\sqrt{k}$ ***buffers***. Each buffer is a FIFO queue that can hold $2k^{3/2}$ elements. Finally, the outputs of the buffers are connected to the $\sqrt{k}$ inputs of the $\sqrt{k}$-merger r in the right part of the figure. The output of this final $\sqrt{k}$-merger becomes the output of the whole k-merger. The reader should notice that the intermediate buffers are overdimensioned. In fact, each buffer can hold $2k^{3/2}$ elements, which is twice the number $k^{3/2}$ of elements output by a $\sqrt{k}$-merger. This additional buffer space is necessary for the correct behavior of the algorithm, as will be explained below. The base case of the recursion is a k-merger with $k = 2$, which produces $k^3 = 8$ elements whenever invoked.

A k-merger operates recursively. In order to output k^3 elements, the k-merger invokes r $k^{3/2}$ times. Before each invocation, however, the k-merger fills all buffers that are less than half full, i.e., all buffers that contain less than $k^{3/2}$ elements. In order to fill buffer i, the algorithm invokes the corresponding left merger l_i once. Since l_i outputs $k^{3/2}$ elements, the buffer contains at least $k^{3/2}$ elements after l_i finishes.

It can be proven by induction that the work complexity of funnelsort is $O(n \lg n)$, which is optimal for comparison-based sorting algorithms [42]. In the rest of this section, we analyze the cache complexity of funnelsort. The goal of the analysis is to show that funnelsort on n elements requires at most $Q(n)$ cache misses, where

$$Q(n) = O\left(1 + (n/L)\left(1 + \log_Z n\right)\right) ,$$

provided that $Z = \Omega(L^2)$.

In order to prove this result, we need three auxiliary lemmas. The first lemma bounds the space required by a k-merger.

Lemma 3 *A k-merger can be laid out in $O(k^2)$ contiguous memory locations.*

Proof: A k-merger requires $O(k^2)$ memory locations for the buffers, plus the space required by the $(\sqrt{k}+1)$ inferior $\sqrt{k}$-mergers. The space $S(k)$ thus satisfies the recurrence

$$S(k) \leq (\sqrt{k}+1)S(\sqrt{k}) + O(k^2) ,$$

whose solution is $S(k) = O(k^2)$. ∎

In order to achieve the bound on $Q(n)$, it is important that the buffers in a k-merger be maintained as circular queues of size k. This requirement guarantees that we can manage the queue cache-efficiently, in the sense stated by the next lemma.

Lemma 4 *Performing r insert and remove operations on a circular queue causes $O(1+r/L)$ cache misses if two cache lines are reserved for the buffer.*

Proof: We reserve the two cache lines to the head and tail of the circular queue. If a new cache line is read during a insert operation, the next $L-1$ insert operations do not cause a cache miss. Consequently, r insert operations incur at most $O(1 + r/L)$ cache misses. The argument for removals is similar. ∎

The next lemma bounds the number of cache misses Q_{M} incurred by a k-merger.

Lemma 5 *If $Z = \Omega(L^2)$, then a k-merger operates with at most $Q_{\mathrm{M}}(k)$ cache misses, where*

$$Q_{\mathrm{M}}(k) = O\left(k + k^3/L + (k^3 \log_Z k)/L\right) .$$

Proof: There are two cases: either $k < \sqrt{\alpha Z}$ or $k > \sqrt{\alpha Z}$, where α is a sufficiently small constant, as usual.

Case I: Assume first that $k < \sqrt{\alpha Z}$.

By Lemma 3, the data structure associated with the k-merger requires at most $O(k^2) = O(\alpha Z)$ contiguous memory locations, and therefore it fits into cache provided that α is small enough. The k-merger has k input queues, from which it loads $O(k^3)$ elements. Let r_i be the number of elements extracted from the ith input queue. Since $k < \sqrt{\alpha Z}$ and $L = O(\sqrt{Z})$, there are at least $Z/L = \Omega(k)$ cache lines available for the input buffers. We assume that the optimal replacement policy reserves these cache lines for the input buffers, so that Lemma 4 applies. This assumption is wlog: We show that this replacement policy achieves the stated bounds, and the optimal policy can only incur fewer cache misses. By Lemma 4, the total number of cache misses for accessing the input queues is

$$\sum_{i=1}^{k} O(1 + r_i/L) = O(k + k^3/L) .$$

Similarly by Lemma 4, the cache complexity of writing the output queue is at most $O(1 + k^3/L)$. Finally, the algorithm incurs at most $O(1 + k^2/L)$ cache misses for touching its internal data structures. The total cache complexity is therefore $Q_{\mathrm{M}}(k) = O\left(k + k^3/L\right)$, completing the proof of the first case.

Case II: Assume now that $k > \sqrt{\alpha Z}$. In this second case, we prove by induction on k that whenever $k > \sqrt{\alpha Z}$, we have

$$Q_{\mathrm{M}}(k) \leq (ck^3 \log_Z k)/L - A(k) , \tag{3.6}$$

for some constant $c > 0$, where $A(k) = k(1 + 2c \log_Z k/L) = o(k^3)$. This particular value of $A(k)$ will be justified later in the analysis.

The base case of the induction consists of values of k such that $(\alpha Z)^{1/4} < k \leq \sqrt{\alpha Z}$. (It is not sufficient to just consider $k = \Theta(\sqrt{Z})$, since k can become as small as $\Theta(Z^{1/4})$ in the recursive calls.) The analysis of the first case applies, yielding $Q_{\mathrm{M}}(k) = O\left(k + k^3/L\right)$. Because $k^2 \geq \sqrt{\alpha Z} = \Omega(L)$ and $k = \Omega(1)$, the last term dominates, and $Q_{\mathrm{M}}(k) = O\left(k^3/L\right)$ holds. Consequently, a large enough value of c can be found that satisfies Inequality (3.6).

For the inductive case, let $k > \sqrt{\alpha Z}$. The k-merger invokes the $\sqrt{k}$-mergers recursively. Since $(\alpha Z)^{1/4} \leq \sqrt{k} \leq k$, the inductive hypothesis can be used to bound the number $Q_{\mathrm{M}}(\sqrt{k})$ of cache misses incurred by the submergers. The "right" merger r is invoked exactly $k^{3/2}$ times. The total number l of invocations of "left" mergers is bounded by $l < k^{3/2} + 2\sqrt{k}$. To see why, consider that every invocation of a left merger puts $k^{3/2}$ elements into some buffer. Since k^3 elements are output and the buffer space is $2k^2$, the bound $l < k^{3/2} + 2\sqrt{k}$ follows.

Before invoking r, the algorithm must check every buffer to see whether it is empty. One such check requires at most $\sqrt{k}$ cache misses, since there are $\sqrt{k}$ buffers. This check is repeated exactly $k^{3/2}$ times, leading to at most k^2 cache misses for all checks.

These considerations lead to the recurrence

$$Q_{\mathrm{M}}(k) \leq \left(2k^{3/2} + 2\sqrt{k}\right) Q_{\mathrm{M}}(\sqrt{k}) + k^2 \ .$$

Application of the inductive hypothesis yields the desired bound Inequality (3.6), as follows:

$$\begin{aligned}
Q_{\mathrm{M}}(k) &\leq \left(2k^{3/2} + 2\sqrt{k}\right) Q_{\mathrm{M}}(\sqrt{k}) + k^2 \\
&\leq 2\left(k^{3/2} + \sqrt{k}\right) \left[\frac{ck^{3/2}\log_Z k}{2L} - A(\sqrt{k})\right] + k^2 \\
&\leq \left(ck^3 \log_Z k\right)/L + k^2\left(1 + (c\log_Z k)/L\right) - \left(2k^{3/2} + 2\sqrt{k}\right) A(\sqrt{k}) \ .
\end{aligned}$$

If $A(k) = k(1 + (2c\log_Z k)/L)$ (for example) Inequality (3.6) follows.

■

Theorem 6 *If $Z = \Omega(L^2)$, then funnelsort sorts n elements with at most $Q(n)$ cache misses, where*

$$Q(n) = O\left(1 + (n/L)\left(1 + \log_Z n\right)\right) \ .$$

Proof: If $n < \alpha Z$ for a small enough constant α, then funnelsort's datastructures fit into cache. To see why, observe that funnelsort invokes only one k-merger at any time. The biggest k-merger is the top-level $n^{1/3}$-merger, which requires $O(n^{2/3}) < O(n)$ space. The algorithm thus can operate in $O(1 + n/L)$ cache misses.

If $N > \alpha Z$, we have the recurrence

$$Q(n) = n^{1/3}Q(n^{2/3}) + Q_{\mathrm{M}}(n^{1/3}) \ .$$

By Lemma 5, we have $Q_{\mathrm{M}}(n^{1/3}) = O\left(n^{1/3} + n/L + n\log_Z n/L\right)$.

With the tall-cache hypothesis $Z = \Omega(L^2)$, we have $n/L = \Omega(n^{1/3})$. Moreover, we also have $n^{1/3} = \Omega(1)$ and $\lg n = \Omega(\lg Z)$. Consequently, $Q_{\mathrm{M}}(n^{1/3}) = O\left((n\log_Z n)/L\right)$ holds, and the recurrence simplifies to

$$Q(n) = n^{1/3}Q(n^{2/3}) + O\left((n\log_Z n)/L\right) \ .$$

The result follows by induction on n. ■

This upper bound matches the lower bound stated by the next theorem, proving that funnelsort is cache-optimal.

Theorem 7 *The cache complexity of any sorting algorithm is*

$$Q(n) = \Omega\left(1 + (n/L)\left(1 + \log_Z n\right)\right) .$$

Proof: Aggarwal and Vitter [6] show that there is an $\Omega\left((n/L)\log_{Z/L}(n/Z)\right)$ bound on the number of cache misses made by any sorting algorithm on their "out-of-core" memory model, a bound that extends to the ideal-cache model. The theorem can be proved by applying the tall-cache assumption $Z = \Omega(L^2)$ and the trivial lower bounds of $Q(n) = \Omega(1)$ and $Q(n) = \Omega(n/L)$. ∎

3.4 Distribution sort

In this section, we describe another cache-oblivious optimal sorting algorithm based on distribution sort. Like the funnelsort algorithm from Section 3.3, the distribution-sorting algorithm uses $O(n \lg n)$ work to sort n elements, and it incurs $O\left(1 + (n/L)\left(1 + \log_Z n\right)\right)$ cache misses if the cache is tall. Unlike previous cache-efficient distribution-sorting algorithms [4, 6, 120, 148, 150], which use sampling or other techniques to find the partitioning elements before the distribution step, our algorithm uses a "bucket splitting" technique to select pivots incrementally during the distribution.

Given an array A (stored in contiguous locations) of length n, the cache-oblivious distribution sort performs sorts A as follows:

1. Partition A into $\sqrt{n}$ contiguous subarrays of size $\sqrt{n}$. Recursively sort each subarray.
2. Distribute the sorted subarrays into q buckets $B_1, \ldots, B_q$ of size $n_1, \ldots, n_q$, respectively, such that
   (a) $\max\{x \mid x \in B_i\} \le \min\{x \mid x \in B_{i+1}\}$ for all $1 \le i < q$;
   (b) $n_i \le 2\sqrt{n}$ for all $1 \le i \le q$.

   (See below for details.)
3. Recursively sort each bucket.
4. Copy the sorted buckets to array A.

A stack-based memory allocator is used to exploit spatial locality.

Distribution step The goal of Step 2 is to distribute the sorted subarrays of A into q buckets $B_1, B_2, \ldots, B_q$. The algorithm maintains two invariants. First, each bucket holds at most $2\sqrt{n}$ elements at any time, and any element in bucket B_i is smaller than any element in bucket B_{i+1}.

Second, every bucket has an associated pivot. Initially, only one empty bucket exists whose pivot is ∞.

The idea is to copy all elements from the subarrays into the buckets while maintaining the invariants. We keep state information for each subarray and bucket. The state of a subarray consists of the index *next* of the next element to be read from the subarray and the bucket number *bnum* where this element should be copied. By convention, $bnum = \infty$ if all elements in a subarray have been copied. The state of a bucket consists of the pivot and the number of elements currently in the bucket.

We would like to copy the element at position *next* of a subarray to bucket *bnum*. If this element is greater than the pivot of bucket *bnum*, we would increment *bnum* until we find a bucket for which the element is smaller than the pivot. Unfortunately, this basic strategy has poor caching behavior, which calls for a more complicated procedure.

The distribution step is accomplished by the recursive procedure DISTRIBUTE(i, j, m) which distributes elements from the ith through $(i + m - 1)$th subarrays into buckets starting from B_j. Given the precondition that each subarray $i, i+1, \ldots, i+m-1$ has its $bnum \geq j$, the execution of DISTRIBUTE(i, j, m) enforces the postcondition that subarrays $i, i+1, \ldots, i+m-1$ have their $bnum \geq j + m$. Step 2 of the distribution sort invokes DISTRIBUTE$(1, 1, \sqrt{n})$. The following is a recursive implementation of DISTRIBUTE:

```
DISTRIBUTE(i, j, m)
if m = 1
  then COPYELEMS(i, j)
  else DISTRIBUTE(i, j, m/2)
       DISTRIBUTE(i + m/2, j, m/2)
       DISTRIBUTE(i, j + m/2, m/2)
       DISTRIBUTE(i + m/2, j + m/2, m/2)
```

In the base case, the procedure COPYELEMS(i, j) copies all elements from subarray i that belong to bucket j. If bucket j has more than $2\sqrt{n}$ elements after the insertion, it can be split into two buckets of size at least $\sqrt{n}$. For the splitting operation, we use the deterministic median-finding algorithm [42, p. 189] followed by a partition. The median-finding algorithm uses $O(m)$ work and incurs $O(1 + m/L)$ cache misses to find the median of an array of size m. (In our case, we have $m = 2\sqrt{n} + 1$.) In addition, when a bucket splits, all subarrays whose *bnum* is greater than the *bnum* of the split bucket must have their *bnum*'s incremented. The analysis of DISTRIBUTE is given by the next two lemmas.

Lemma 8 *The median of n elements can be found cache-obliviously using $O(n)$ work and incurring $O(1 + n/L)$ cache misses.*

Proof: See [42, p. 189] for the linear-time median finding algorithm and the work analysis. The cache complexity is given by the same recurrence as the work complexity with a different base case.

$$Q(m) = \begin{cases} O(1 + m/L) & \text{if } m \leq \alpha Z \ , \\ Q(\lceil m/5 \rceil) + Q(7m/10 + 6) & \\ \qquad\qquad\qquad + O(1 + m/L) & \text{otherwise} \ , \end{cases}$$

where α is a sufficiently small constant. The result follows. ∎

Lemma 9 *The distribute step uses $O(n)$ work, incurs $O(1 + n/L)$ cache misses, and uses $O(n)$ stack space to distribute n elements.*

Proof: In order to simplify the analysis of the work used by DISTRIBUTE, assume that COPY-ELEMS uses $O(1)$ work for procedural overhead. We account for the work due to copying elements and splitting of buckets separately. The work of DISTRIBUTE is described by the recurrence

$$T(c) = 4T(c/2) + O(1) \ .$$

It follows that $T(c) = O(c^2)$, where $c = \sqrt{n}$ initially. The work due to copying elements is also $O(n)$.

The total number of bucket splits is at most $\sqrt{n}$. To see why, observe that there are at most $\sqrt{n}$ buckets at the end of the distribution step, since each bucket contains at least $\sqrt{n}$ elements. Each split operation involves $O(\sqrt{n})$ work and so the net contribution to the work is $O(n)$. Thus, the total work used by DISTRIBUTE is $W(n) = O(T(\sqrt{n})) + O(n) + O(n) = O(n)$.

For the cache analysis, we distinguish two cases. Let α be a sufficiently small constant such that the stack space used fits into cache.

Case I: $n \leq \alpha Z$.

The input and the auxiliary space of size $O(n)$ fit into cache using $O(1 + n/L)$ cache lines. Consequently, the cache complexity is $O(1 + n/L)$.

Case II: $n > \alpha Z$.

Let $R(c, m)$ denote the cache misses incurred by an invocation of DISTRIBUTE(a, b, c) that copies m elements from subarrays to buckets. We again account for the splitting of buckets separately. We first prove that R satisfies the following recurrence:

$$R(c, m) \leq \begin{cases} O(L + m/L) & \text{if } c \leq \alpha L \ , \\ \sum_{1 \leq i \leq 4} R(c/2, m_i) & \text{otherwise} \ , \end{cases} \tag{3.7}$$

where $\sum_{1 \le i \le 4} m_i = m$.

First, consider the base case $c \le \alpha L$. An invocation of DISTRIBUTE(a, b, c) operates with c subarrays and c buckets. Since there are $\Omega(L)$ cache lines, the cache can hold all the auxiliary storage involved and the currently accessed element in each subarray and bucket. In this case there are $O(L + m/L)$ cache misses. The initial access to each subarray and bucket causes $O(c) = O(L)$ cache misses. The cache complexity of copying the m elements from contiguous to contiguous locations is $O(1+m/L)$. This completes the proof of the base case. The recursive case, when $c > \alpha L$, follows immediately from the algorithm. The solution for Equation (3.7) is $R(c, m) = O(L + c^2/L + m/L)$.

We still need to account for the cache misses caused by the splitting of buckets. Each split causes $O(1 + \sqrt{n}/L)$ cache misses due to median finding (Lemma 8) and partitioning of $\sqrt{n}$ contiguous elements. An additional $O(1+\sqrt{n}/L)$ misses are incurred by restoring the cache. As proven in the work analysis, there are at most $\sqrt{n}$ split operations.

By adding $R(\sqrt{n}, n)$ to the split complexity, we conclude that the total cache complexity of the distribution step is $O(L + n/L + \sqrt{n}(1 + \sqrt{n}/L)) = O(n/L)$.

■

Theorem 10 *Distribution sort uses $O(n \lg n)$ work and incurs $O(1 + (n/L)\,(1 + \log_Z n))$ cache misses to sort n elements.*

Proof: The work done by the algorithm is given by

$$W(n) = \sqrt{n}W(\sqrt{n}) + \sum_{i=1}^{q} W(n_i) + O(n) ,$$

where each $n_i \le 2\sqrt{n}$ and $\sum n_i = n$. The solution to this recurrence is $W(n) = O(n \lg n)$.

The space complexity of the algorithm is given by

$$S(n) \le S(2\sqrt{n}) + O(n) ,$$

where the $O(n)$ term comes from Step 2. The solution to this recurrence is $S(n) = O(n)$.

The cache complexity of distribution sort is described by the recurrence

$$Q(n) \le \begin{cases} O(1 + n/L) & \text{if } n \le \alpha Z \text{ ,} \\ \sqrt{n}Q(\sqrt{n}) + \sum_{i=1}^{q} Q(n_i) & \text{otherwise ,} \\ \qquad\qquad +O(1 + n/L) & \end{cases}$$

where α is a sufficiently small constant such that the stack space used by a sorting problem of size αZ, including the input array, fits completely in cache. The base case $n \leq \alpha Z$ arises when both the input array A and the contiguous stack space of size $S(n) = O(n)$ fit in $O(1 + n/L)$ cache lines of the cache. In this case, the algorithm incurs $O(1 + n/L)$ cache misses to touch all involved memory locations once. In the case where $n > \alpha Z$, the recursive calls in Steps 1 and 3 cause $Q(\sqrt{n}) + \sum_{i=1}^{q} Q(n_i)$ cache misses and $O(1 + n/L)$ is the cache complexity of Steps 2 and 4, as shown by Lemma 9. The theorem now follows by solving the recurrence. ∎

3.5 Other cache models

In this section we show that cache-oblivious algorithms designed in the two-level ideal-cache model can be efficiently ported to other cache models. We show that algorithms whose complexity bounds satisfy a simple regularity condition (including all algorithms heretofore presented) can be ported to less-ideal caches incorporating least-recently-used (LRU) or first-in, first-out (FIFO) replacement policies [79, p. 378]. We argue that optimal cache-oblivious algorithms are also optimal for multilevel caches. Finally, we present simulation results proving that optimal cache-oblivious algorithms satisfying the regularity condition are also optimal (in expectation) in the previously studied SUMH [11, 148] and HMM [4] models. Thus, all the algorithmic results in this chapter apply to these models, matching the best bounds previously achieved.

3.5.1 Two-level models

Many researchers, such as [6, 82, 149], employ two-level models similar to the ideal-cache model, but without an automatic replacement strategy. In these models, data must be moved explicitly between the the primary and secondary levels "by hand." We define a cache complexity bound $Q(n; Z, L)$ to be ***regular*** if

$$Q(n; Z, L) = O(Q(n; 2Z, L)) \ . \tag{3.8}$$

We now show that optimal algorithms in the ideal-cache model whose cache complexity bounds are regular can be ported to these models to run using optimal work and incurring an optimal expected number of cache misses.

The first lemma shows that the optimal and omniscient replacement strategy used by an ideal cache can be simulated efficiently by the LRU and FIFO replacement strategies.

Lemma 11 *Consider an algorithm that causes $Q^*(n; Z, L)$ cache misses on a problem of size n using a (Z, L) ideal cache. Then, the same algorithm incurs $Q(n; Z, L) \leq 2Q^*(n; Z/2, L)$ cache misses on a (Z, L) cache that uses either LRU or FIFO replacement.*

Proof: Sleator and Tarjan [133] have shown that the cache misses on a (Z, L) cache using LRU replacement is $(Z/(Z - Z^* + 1))$-competitive with optimal replacement on a (Z^*, L) ideal if both caches start with an empty cache. It follows that the number of misses on a (Z, L) LRU-cache is at most twice the number of misses on a $(Z/2, L)$ ideal-cache. The same argument holds for FIFO caches. ■

Corollary 12 *For algorithms with regular cache complexity bounds, the asymptotic number of cache misses is the same for LRU, FIFO, and optimal replacement.*

Proof: Follows directly from Lemma 11 and the regularity condition Equation (3.8). ■

Since previous two-level models do not support automatic replacement, to port a cache-oblivious algorithms to them, we implement a LRU (or FIFO) replacement strategy in software.

Lemma 13 *A (Z, L) LRU-cache (or FIFO-cache) can be maintained using $O(Z)$ primary memory locations such that every access to a cache line in primary memory takes $O(1)$ expected time.*

Proof: Given the address of the memory location to be accessed, we use a 2-universal hash function [114, p. 216] to maintain a hash table of cache lines present in the primary memory. The Z/L entries in the hash table point to linked lists in a heap of memory containing Z/L records corresponding to the cache lines. The 2-universal hash function guarantees that the expected size of a chain is $O(1)$. All records in the heap are organized as a doubly linked list in the LRU order (or singly linked for FIFO). Thus, the LRU (FIFO) replacement policy can be implemented in $O(1)$ expected time using $O(Z/L)$ records of $O(L)$ words each. ■

Theorem 14 *An optimal cache-oblivious algorithm with a regular cache-complexity bound can be implemented optimally in expectation in two-level models with explicit memory management.*

Proof: Follows from Corollary 12 and Lemma 13. ■

Consequently, our cache-oblivious algorithms for matrix multiplication, matrix transpose, FFT, and sorting are optimal in two-level models.

3.5.2 Multilevel ideal caches

We now show that optimal cache-oblivious algorithms also perform optimally in computers with multiple levels of ideal caches. Moreover, Theorem 14 extends to multilevel models with explicit memory management.

The $\langle (Z_1, L_1), (Z_2, L_2), \ldots, (Z_r, L_r) \rangle$ ***ideal-cache model*** consists of an arbitrarily large main memory and a hierarchy of r caches, each of which is managed by an optimal replacement strategy. The model assumes that the caches satisfy the ***inclusion property*** [79, p. 723], which says that for $i = 1, 2, \ldots, r-1$, the values stored in cache i are also stored in cache $i+1$. The performance of an algorithm running on an input of size n is measured by its work complexity $W(n)$ and its cache complexities $Q_i(n; Z_i, L_i)$ for each level $i = 1, 2, \ldots, r$.

Theorem 15 *An optimal cache-oblivious algorithm in the ideal-cache model incurs an asymptotically optimal number of cache misses on each level of a multilevel cache with optimal replacement.*

Proof: The theorem follows directly from the definition of cache obliviousness and the optimality of the algorithm in the two-level ideal-cache model. ■

Theorem 16 *An optimal cache-oblivious algorithm with a regular cache-complexity bound incurs an asymptotically optimal number of cache misses on each level of a multilevel cache with LRU, FIFO, or optimal replacement.*

Proof: Follows from Corollary 12 and Theorem 16. ■

3.5.3 The SUMH model

In 1990 Alpern et al. [11] presented the uniform memory hierarchy model (UMH), a parameterized model for a memory hierarchy. In the $\text{UMH}_{\alpha,\rho,b(l)}$ model, for integer constants $\alpha, \rho > 1$, the size of the ith memory level is $Z_i = \alpha\rho^{2i}$ and the line length is $L_i = \rho^i$. A transfer of one ρ^l-length line between the caches on level l and $l+1$ takes $\rho^l/b(l)$ time. The bandwidth function $b(l)$ must be nonincreasing and the processor accesses the cache on level 1 in constant time per access. An algorithm given for the UMH model must include a schedule that, given for a particular set of input variables, tells exactly when each block is moved along which of the buses between caches. Work and cache misses are folded into one cost measure $T(n)$. Alpern et al. prove that an algorithm that performs the optimal number of I/O's at all levels of the hierarchy does not necessarily run in optimal time in the UMH model, since scheduling bottlenecks can occur when all buses are active. In the more restrictive SUMH model [148], however, only one bus is active at a time. Consequently, we can prove that optimal cache-oblivious algorithms run in optimal expected time in the SUMH model.

Lemma 17 *A cache-oblivious algorithm with $W(n)$ work and $Q(n; Z, L)$ cache misses on a (Z, L)-ideal cache can be executed in the $SUMH_{\alpha,\rho,b(l)}$ model in expected time*

$$T(n) = O\Big(W(n) + \sum_{i=1}^{r-1} \frac{\rho^i}{b(i)} Q(n; \Theta(Z_i), L_i)\Big) ,$$

where $Z_i = \alpha\rho^{2i}$, $L_i = \rho^i$, and Z_r is big enough to hold all elements used during the execution of the algorithm.

Proof: Use the memory at the ith level as a cache of size $Z_i = \alpha\rho^{2i}$ with line length $L_i = \rho^i$ and manage it with software LRU described in Lemma 13. The rth level is the main memory, which is direct mapped and not organized by the software LRU mechanism. An LRU-cache of size $\Theta(Z_i)$ can be simulated by the ith level, since it has size Z_i. Thus, the number of cache misses at level i is $2Q(n; \Theta(Z_i), L_i)$, and each takes $\rho^i/b(i)$ time. Since only one memory movement happens at any point in time, and there are $O(W(n))$ accesses to level 1, the lemma follows by summing the individual costs. ∎

Lemma 18 *Consider a cache-oblivious algorithm whose work on a problem of size n is lower-bounded by $W^*(n)$ and whose cache complexity is lower-bounded by $Q^*(n; Z, L)$ on an (Z, L) ideal-cache. Then, no matter how data movement is implemented in SUMH$_{\alpha,\rho,b(l)}$, the time taken on a problem of size n is at least*

$$T(n) = \Omega\Big(W^*(n) + \sum_{i=1}^{r} \frac{\rho^i}{b(i)} Q^*(n, \Theta(Z_j), L_i)\Big) ,$$

where $Z_i = \alpha\rho^{2i}$, $L_i = \rho^i$ and Z_r is big enough to hold all elements used during the execution of the algorithm.

Proof: The optimal scheduling of the data movements does not need to obey the inclusion property, and thus the number of ith-level cache misses is at least as large as for an ideal cache of size $\sum_{j=1}^{i} Z_i = O(Z_i)$. Since $Q^*(n, Z, L)$ lower-bounds the cache misses on a cache of size Z, at least $Q^*(n, \Theta(Z_i), L_i)$ data movements occur at level i, each of which takes $\rho^i/b(i)$ time. Since only one movement can occur at a time, the total cost is the maximum of the work and the sum of the costs at all the levels, which is within a factor of 2 of their sum. ∎

Theorem 19 *A cache-oblivious algorithm that is optimal in the ideal-cache model and whose cache-complexity is regular can be executed optimal expected time in the SUMH$_{\alpha,\rho,b(l)}$ model.*

Proof: The theorem follows directly from regularity and Lemmas 17 and 18. ∎

3.6 Related work

In this section, we discuss the origin of the notion of cache-obliviousness. We also give an overview of other hierarchical memory models.

Our research group at MIT noticed as far back as 1994 that divide-and-conquer matrix multiplication was a cache-optimal algorithm that required no tuning, but we did not adopt the term "cache-oblivious" until 1997. This matrix-multiplication algorithm, as well as a cache-oblivious algorithm for LU-decomposition without pivoting, eventually appeared in [26]. Shortly after leaving our research group, Toledo [143] independently proposed a cache-oblivious algorithm for LU-decomposition, but with pivoting. For $n \times n$ matrices, Toledo's algorithm uses $\Theta(n^3)$ work and incurs $\Theta(1 + n^2/L + n^3/L\sqrt{Z})$ cache misses. My own FFTW Fourier transform library employs a register-allocation and scheduling algorithm inspired by the cache-oblivious FFT algorithm. The general idea that divide-and-conquer enhances memory locality has been known for a long time [132].

Previous theoretical work on understanding hierarchical memories and the I/O-complexity of algorithms has been studied in cache-aware models lacking an automatic replacement strategy. Hong and Kung [82] use the red-blue pebble game to prove lower bounds on the I/O-complexity of matrix multiplication, FFT, and other problems. The red-blue pebble game models temporal locality using two levels of memory. The model was extended by Savage [129] for deeper memory hierarchies. Aggarwal and Vitter [6] introduced spatial locality and investigated a two-level memory in which a block of P contiguous items can be transferred in one step. They obtained tight bounds for matrix multiplication, FFT, sorting, and other problems. The hierarchical memory model (HMM) by Aggarwal et al. [4] treats memory as a linear array, where the cost of an access to element at location x is given by a cost function $f(x)$. The BT model [5] extends HMM to support block transfers. The UMH model by Alpern et al. [11] is a multilevel model that allows I/O at different levels to proceed in parallel. Vitter and Shriver introduce parallelism, and they give algorithms for matrix multiplication, FFT, sorting, and other problems in both a two-level model [149] and several parallel hierarchical memory models [150]. Vitter [147] provides a comprehensive survey of external-memory algorithms.

3.7 Conclusion

In this chapter, we discussed the notion of cache-obliviousness, and we presented optimal cache-oblivious algorithms for rectangular matrix transpose and multiplication, FFT, and sorting. Cache-oblivious algorithms are inherently portable, because they depend on no tuning parameters, and optimal cache-oblivious algorithms enable portability of performance across systems with diverse memory hierarchies. We learned that divide and conquer can yield algorithms that are good from both Cilk's perspective, because they have short critical path, and from the point of view of the memory hierarchy, because they achieve the optimal cache complexity.

Far from answering all questions in portable high performance, however, this chapter open more problems than I am capable of solving. Intuitively, I would expect the cache complexity of cache-aware algorithms to be inherently lower than the complexity of cache-oblivious algorithms, but the results of this chapter contradict this intuition. Do optimal cache-oblivious algorithms exist for all problems, or can we find a problem for which cache-aware algorithms are inherently better? This problem is open for future research.

A second set of questions arises when we try to run a cache-oblivious algorithm in parallel, for example using Cilk. Running these algorithms in parallel would produce a formidable combination of portability and high performance, because the resulting program would be high-performance and yet insensitive to both the number of processors and the memory hierarchy. Unfortunately, things are not so easy. The analysis Cilk scheduler offers no performance guarantees if Cilk threads are delayed by cache misses, and conversely, the analysis of cache-oblivious algorithm offer no cache-complexity guarantees in a Cilk environment where the scheduler moves threads across the parallel machine. The problem of combining Cilk with cache-oblivious algorithms is not completely open, however, and we shall discuss a possible solution in Chapter 4.

The ideal-cache model is not an adequate model of write-through caches. In many modern processor, the L1 cache is ***write-through***, i.e., it transmits written values to the L2 cache immediately. With write-through caches, we can no longer argue that once a problem fits into cache no further misses are incurred, since the cache incurs a “miss” at every write operation. We currently do not know how to account for write-through caches in our theory of cache-oblivious algorithms.

Chapter 4

Portable parallel memory

In this chapter we attempt to marry Cilk with cache-oblivious algorithms. In Cilk, we can write high-performance programs that run efficiently with varying degrees of parallelism. The theory of cache-oblivious algorithms allows us to design fast algorithms that are insensitive to the parameters of the memory hierarchy. What happens when we code the cache-oblivious algorithms in Cilk and run them on a parallel machine? Specifically, consider the following two questions.

1. Can we preserve Cilk's performance guarantees and its empirical efficiency if we augment the Cilk scheduler with a cache? The Cilk theory of Section 2.3 does not mention caches at all. The execution-time upper bound from [25] does not hold in the presence of caches, because the proof does not account for the time spent in servicing cache misses.

2. Is the cache complexity preserved when a program is executed in parallel? For example, if work is moved from one processor to another, the contents of the first cache are unavailable to the destination processor, and communication between caches is necessary for the correct execution of the program.

The answer to these two questions seems to depend crucially on the memory model that we use. A ***memory model*** is a specification of how memory behaves in a computer system. To see why a good memory model is important, imagine executing a Cilk program on a network of workstations in which each processor operates within its own memory and no attempt is ever made to synchronize the memory contents. Such a system would be very fast, since workstations do not communicate at all, but most likely useless since processors cannot see each other's results. On the other extreme, the ***sequential consistency*** model [96] dictates that the whole memory of the machine behave as a single black box, so that every processor sees the same order of memory events (reads and writes). Sequential consistency appears at first sight to be the ideal memory model, because it preserves

This chapter represents joint work with Bobby Blumofe, Chris Joerg, and Charles Leiserson, and Keith Randall. A preliminary version appears in [27, 26].

the black-box abstraction of a single memory, but unfortunately, sequential consistency has a price. It is generally believed [79] that sequential consistency imposes major inefficiencies in an implementation. (See [81] for the opposite view, however.) Consequently, many researchers have tried to relax the requirements of sequential consistency in exchange for better performance and ease of implementation. For example, ***processor consistency*** [70] is a model where every processor can have an independent view of memory, and ***release consistency*** [64] is a model where the memory becomes consistent only when certain synchronizing operations are performed. See [1] for a good tutorial on this subject.

In this chapter, we focus on a memory model called ***location consistency***.[1] Location consistency is relevant to portable high performance because it is the memory model maintained by the **BACKER** coherence algorithm, and a combination of BACKER and Cilk executes a cache-oblivious Cilk program maintaining both the performance guarantees of Cilk and the program's cache complexity. Specifically, we prove that a Cilk program with work T_1, critical path T_∞, and cache-complexity $Q(Z, L)$ runs on P processors in expected time

$$T_P = O((T_1 + \mu Q(Z, L))/P + \mu Z T_\infty / L) ,$$

where μ is the cost of transferring one cache line between main memory and the cache. To my knowledge, the combination of Cilk and **BACKER** is the only shared-memory programming system algorithm with any sort of performance guarantee. While the BACKER coherence algorithm is simplistic and does not attempt optimizations, it has been implemented in the Cilk-3 runtime system with encouraging empirical results [27].

To illustrate the concepts behind location consistency, consider again the `matrixmul` program from Section 2.4. Like any Cilk multithreaded computation [28], the parallel instruction stream of `matrixmul` can be viewed as a "spawn tree" of procedures broken into a directed acyclic graph, or ***dag***, of "threads." The ***spawn tree*** is exactly analogous to a traditional call tree. When a procedure, such as `matrixmul` performs a spawn, the spawned procedure becomes a child of the procedure that performed the spawn. Each procedure is broken by `sync` statements into nonblocking sequences of instructions, called ***threads***, and the threads of the computation are organized into a dag representing the partial execution order defined by the program. Figure 4-1 illustrates the structure of the dag for `matrixmul`. Each vertex corresponds to a thread of the computation, and the edges define the partial execution order. The syncs in lines 21 and 23 break the procedure `matrixmul` into three threads u, v, and w, which correspond respectively to the partitioning and spawning of subproblems $M_0, M_1, \ldots, M_7$ in lines 2–20, the spawning of the addition S in line 22, and the return in line 25.

[1] Location consistency is often called coherence in the literature [79]. It is *not* the model with the same name introduced by Gao and Sarkar [61]. See [54] for a justification of this terminology.

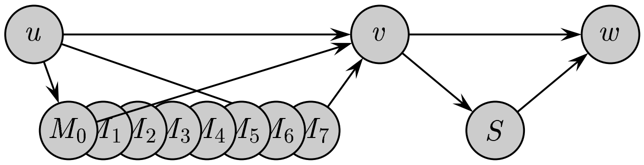


Figure 4-1: Dag generated by the execution of the matrix multiplication program in Figure 2-4. Some edges have been omitted for clarity.

Location-consistent shared memory is a natural consistency model to support a shared-memory program such as `matrixmul`. Certainly, sequential consistency [96] can guarantee the correctness of the program, but a closer look at the precedence relation given by the dag reveals that a much weaker consistency model suffices. Specifically, the 8 recursively spawned children $M_0, M_1, \ldots, M_7$ need not have the same view of shared memory, because the portion of shared memory that each writes is neither read nor written by the others. On the other hand, the parallel addition of `tmp` into `R` by the computation S requires S to have a view in which all of the writes to shared memory by $M_0, M_1, \ldots, M_7$ have completed.

The intuition behind location consistency is that each memory location sees values that are consistent with some serial execution order of the dag, but two different locations may see different serial orders. Thus, the writes performed by a thread are seen by its successors, but threads that are incomparable in the dag may or may not see each other's writes. In `matrixmul`, the computation S sees the writes of $M_0, M_1, \ldots, M_7$, because all the threads of S are successors of $M_0, M_1, \ldots, M_7$, but since the M_i are incomparable, they cannot depend on seeing each others writes. We shall define location consistency precisely in Section 4.2.

All threads of a multithreaded computation should have access to a single, shared virtual address space, and in order to support such a shared-memory abstraction on a computer with physically distributed memory, the runtime scheduler must be coupled with a coherence algorithm. For our BACKER coherence algorithm, we assume that each processor's memory is divided into two regions, each containing lines of shared-memory objects. One region is a ***cache*** of size Z, partitioned into Z/L lines of length L containing locations that have been recently accessed by that processor. The rest of each processors' memory is maintained as a ***main memory*** of locations that have been allocated in the virtual address space. Each allocated line is assigned to the main memory of a processor chosen by hashing the cache line's virtual address. In order for a processor to operate on a location, the location must be resident in the processor's cache; otherwise, a cache miss occurs, and BACKER must "fetch" the correct cache line from main memory into the cache. We assume that when a cache miss occurs, no progress can be made on the computation during the time it takes to service the miss, and the miss time may vary due to congestion of concurrent accesses to the main memory. Like in the ideal-cache model of Chapter 3, we shall further assume that lines in the cache are maintained using the LRU (least-recently-used) [88] heuristic. In addition to servicing cache misses, BACKER must "reconcile" cache lines between the processor caches and the main memory so that the semantics of the execution obey the assumptions of location consistency.

The remainder of this chapter is organized as follows. Section 4.1 combines the Cilk performance model and the ideal-cache model, and states the performance of BACKER precisely. Section 4.2 gives a precise definition of location consistency and describes the BACKER coherence algorithm. Section 4.3 analyzes the execution time of fully strict [25] multithreaded algorithms when the execution is scheduled by the randomized work-stealing scheduler and location consistency is maintained by the BACKER coherence algorithm. Section 4.4 analyzes the space requirements of parallel divide-and-conquer algorithms. Finally, Section 4.5 offers some comparisons with other consistency models.

4.1 Performance model and summary of results

This section defines performance measures for location-consistent Cilk programs, and states the main results of this chapter formally. We define the ***total work*** $T_1(Z, L)$ as the serial execution time on a machine with a (Z, L) cache, and we clarify the meaning of critical-path length in programs that use shared memory. We state bounds on the execution time and cache misses of fully strict [25] programs executed by Cilk in conjunction with the BACKER coherence algorithm. We state bounds on the space requirements of parallel divide-and-conquer algorithms. As an example of application, we apply these results to the cache-oblivious Cilk program `matrixmul`.

In order to model the performance of multithreaded algorithms that use location-consistent shared memory, it is important to observe that running times will vary as a function of the cache size Z and of the line size L, and consequently we must introduce measures that account for this dependence. Consider a (Z, L) cache, which contains $H = Z/L$ lines of size L. We call quantity H the ***cache height***. Let μ be the time to service a cache miss in the serial execution. For example, μ might be proportional to the line size L, but here we do not assume any specific relationship between μ and L.

Consider again the multithreaded computation (such as the one in Figure 4-1) that results when a given multithreaded algorithm is used to solve a given problem. We shall define a new work measure, the "total work," that accounts for the cost of cache misses in the serial execution of the computation, as follows. We associate a weight with each instruction of the dag. Each instruction that generates a cache miss in the one-processor execution with the standard, depth-first serial execution order has weight $\mu + 1$, and all other instructions have weight 1. The ***total work***, denoted $T_1(Z, L)$, is the total weight of all instructions in the dag, which corresponds to the serial execution time if cache misses take μ units of time to be serviced. We shall continue to let T_1 denote the number of instructions in the dag, but for clarity, we shall refer to T_1 as the ***computational work***. (The computational work T_1 corresponds to the serial execution time if all cache misses take zero time to be serviced.) To relate these measures, we define the ***serial cache complexity***, denoted $Q(Z, L)$, to be the number of cache misses taken in the serial execution (that is, the number of instructions with weight $\mu + 1$). This measure is the same as the cache complexity of Chapter 3. Thus, we have $T_1(Z, L) = T_1 + \mu Q(Z, L)$. The total work therefore translates both the work and the cache complexity of Chapter 3 into units of execution time. This definition is useful because from the point of view of the Cilk scheduler it does not matter whether threads spend time in computational work or in waiting for cache misses.

The quantity $T_1(Z, L)$ is an unusual measure. Unlike T_1, it depends on the serial execution order of the computation. The quantity $T_1(Z, L)$ further differs from T_1 in that $T_1(Z, L)/P$ is not a lower bound on the execution time for P processors. It is possible to construct a computation containing P subcomputations that run on P separate processors in which each processor repeatedly accesses H different cache lines in sequence. Consequently, with (Z, L) caches, no processor ever misses, except to warm up the cache at the start of the computation. If we run the same computation serially with a cache of height H (or any size less than HP), however, the necessary multiplexing among tasks can cause numerous cache misses. Consequently, for this computation, the execution time with P processors is much less than $T_1(Z, L)/P$. In this dissertation, we shall forgo the possibility of obtaining such superlinear speedup on computations. Instead, we shall simply attempt to obtain linear speedup.

Critical-path length can likewise be split into two notions. We define the ***total critical-path length***, denoted $T_\infty(Z, L)$, to be the maximum over all directed paths in the computational dag, of the time, including cache misses, to execute along the path by a single processor with an (Z, L) cache. The ***computational critical-path length*** T_∞ is the same, but where misses cost zero time. Both T_∞ and $T_\infty(Z, L)$ are lower bounds on execution time. Although $T_\infty(Z, L)$ is the stronger lower bound, it appears difficult to compute and analyze, and our upper-bound results will be characterized in terms of T_∞, which we shall continue to refer to simply as the critical-path length.

The main result of this chapter is the analysis of the execution time of "fully strict" multithreaded algorithms that use location consistent shared memory. A multithreaded computation is ***fully strict*** [25] if every dependency edge goes from a procedure to either itself or its parent procedure. All Cilk-5 computations are fully strict, because a Cilk procedure can return a value only to its parent, but not to its other ancestors. (This constraint is enforced by the call/return semantics of Cilk.) Consequently, the analysis applies to all Cilk programs. The multithreaded algorithm is executed on a parallel computer with P processors, each with a (Z, L) cache, and a cache miss that encounters no congestion is serviced in μ units of time. The execution is scheduled by the Cilk work-stealing scheduler and location consistency is maintained by the BACKER coherence algorithm. In addition, we assume that accesses to shared memory are distributed uniformly and independently over the main memory—often a plausible assumption, since BACKER hashes cache lines to the main memory. The following theorem bounds the parallel execution time.

Theorem 20 *Consider any fully strict multithreaded computation executed on P processors, each with an LRU cache of height H, using the Cilk work-stealing scheduler in conjunction with the* BACKER *coherence algorithm. Let μ be the service time for a cache miss that encounters no congestion, and assume that accesses to the main memory are random and independent. Suppose the computation has T_1 computational work, $Q(Z, L)$ serial cache misses, $T_1(Z, L) = T_1 + \mu Q(Z, L)$ total work, and T_∞ critical-path length. Then for any $\epsilon > 0$, the execution time is $O(T_1(Z, L)/P + \mu H T_\infty + \mu P \lg P + \mu H \lg(1/\epsilon))$ with probability at least $1 - \epsilon$. Moreover, the expected execution time is $O(T_1(Z, L)/P + \mu H T_\infty)$.*

Proof: See Section 4.3. ∎

This theorem enables us to design high-performance portable programs by designing algorithms with optimal work, critical path, and cache complexity. In the cases where we cannot optimize all three quantities simultaneously, Theorem 20 gives a model to investigate the tradeoffs. For example, the critical path of `matrixmul` is $\Theta(\lg^2 n)$. We could write a matrix multiplication program with critical path $\Theta(\lg n)$ by spawning a separate thread to compute each element of the output array, where each thread spawns a divide-and-conquer addition. This algorithm would have a $\Theta(n^3)$ cache complexity, however, while `matrixmul`'s complexity is $\Theta(n^3)/(L\sqrt{Z})$. For large values of n, Theorem 20 predicts that `matrixmul` is faster.

Theorem 20 is not as strong a result as we would like to prove, because accesses to the main memory are not necessarily independent. For example, threads may concurrently access the same cache lines by algorithm design. We can artificially solve this problem by insisting, as does the EREW-PRAM model, that the algorithm performs exclusive accesses only. More seriously, however, congestion delay in accessing the main memory can cause the computation to be scheduled differently than if there were no congestion, thereby perhaps causing more congestion to occur. It may be possible to prove our bounds for a hashed main memory without making this independence assumption, but we do not know how at this time. The problem with independence does not seem to be serious in practice, and indeed, given the randomized nature of our scheduler, it is hard to conceive of how an adversary can actually take advantage of the lack of independence implied by hashing to slow the execution. Although our results are imperfect, we are actually analyzing the effects of congestion, and thus our results are much stronger than if we had assumed, for example, that accesses to the main memory independently suffer Poisson-distributed delays.

In this chapter, we also analyze the number of cache misses that occur during algorithm execution. This is the parallel analogue of the cache complexity. Again, execution is scheduled with the Cilk work-stealing scheduler and location consistency is maintained by the BACKER coherence algorithm, and we assume that accesses to main memory are random and independent. A bound on the number of cache misses is stated by the next corollary.

Corollary 21 *Consider any fully strict multithreaded computation executed on P processors, each with an LRU cache of height H, using the Cilk work-stealing scheduler in conjunction with the* BACKER *coherence algorithm. Assume that accesses to the main memory are random and independent. Suppose the computation has $Q(Z, L)$ serial cache misses and T_∞ critical-path length. Then for any $\epsilon > 0$, the number of cache misses is at most $Q(Z, L) + O(HPT_\infty + HP \lg(1/\epsilon))$ with probability at least $1 - \epsilon$. Moreover, the expected number of cache misses is at most $Q(Z, L) + O(HPT_\infty)$.*

Proof: See Section 4.3. ■

For example, the total number of cache misses incurred by `matrixmul` when multiplying $n \times n$ matrices using P processors is $O(1 + n^2/L + n^3/(L\sqrt{Z}) + HP \lg^2 n)$, assuming that the independence assumption for the main memory holds.

Space utilization of Cilk programs is relevant to portable high performance, too. If a program exhausts memory when run in parallel, it is not portable no matter how fast it is. In this chapter, we analyze the space requirements of "simple" multithreaded algorithms that use location-consistent shared memory. We assume that the computation is scheduled by a scheduler, such as the work-stealing algorithm, that maintains the "busy-leaves" property [25, 30]. For a given simple multithreaded algorithm, let S_1 denote the space required by the standard, depth-first serial execution of the algorithm to solve a given problem. In previous work, Blumofe has shown that the space used by a P-processor execution is at most $S_1 P$ in the worst case [25, 30]. We improve this characterization of the space requirements, and we provide a much stronger upper bound on the space requirements of "regular" divide-and-conquer multithreaded algorithms, in which each thread divides a problem of size n into a subproblems, each of size n/b for some constants $a \geq 1$ and $b > 1$, and then it recursively spawns child threads to solve each subproblem.

Theorem 22 *Consider any regular divide-and-conquer multithreaded algorithm executed on P processors using a busy-leaves scheduler. Suppose that each thread, when spawned to solve a problem of size n, allocates $s(n)$ space, and if n is larger than some constant, then the thread divides the problem into a subproblems each of size n/b for some constants $a \geq 1$ and $b > 1$. Then, the total amount $S_P(n)$ of space taken by the algorithm in the worst case when solving a problem of size n can be determined as follows:*[2]

1. *If $s(n) = \Theta(\lg^k n)$ for some constant $k \geq 0$, then $S_P(n) = \Theta(P \lg^{k+1}(n/P))$.*

[2] *Other cases exist besides those given here.*

2. *If $s(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $S_P(n) = \Theta(Ps(n/P^{1/\log_b a}))$, if, in addition, $s(n)$ satisfies the regularity condition $\gamma_1 s(n/b) \leq s(n) \leq a\gamma_2 s(n/b)$ for some constants $\gamma_1 > 1$ and $\gamma_2 < 1$.*

3. *If $s(n) = \Theta(n^{\log_b a})$, then $S_P(n) = \Theta(s(n) \lg P)$.*

4. *If $s(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, then $S_P(n) = \Theta(s(n))$, if, in addition, $s(n)$ satisfies the regularity condition that $s(n) \geq a\gamma s(n/b)$ for some constant $\gamma > 1$.*

Proof: See Section 4.4. ∎

For example, Theorem 22 applies to `matrixmul` with $a = 8$, $b = 2$, and $s(n) = O(n^2)$. From Case 2, we see that multiplying $n \times n$ matrices on P processors uses only $\Theta(n^2 P^{1/3})$ space, which is tighter than the $O(n^2 P)$ result obtained by directly applying the $S_1 P$ bound.

4.2 Location consistency and the BACKER coherence algorithm

In this section we give a precise definition of location consistency, and we describe the BACKER [27] coherence algorithm for maintaining this memory model. Location consistency is a relaxed consistency model for distributed shared memory, and the BACKER algorithm can maintain location consistency for multithreaded computations that execute on a parallel computer with physically distributed memory. In this chapter we give a simplified definition of location consistency. Chapter 5 offers an equivalent definition (Definition 48) in the more formal ***computation-centric*** theory of memory models.

Shared memory consists of a set of ***locations*** that instructions can read and write. When an instruction performs a read of a location, it receives some value, but the particular value it receives depends upon the consistency model. As its name suggests, location consistency is defined separately for each location in shared memory.

Definition 23 *Let C be the dag of a multithreaded computation. The shared memory $\mathcal{M}$ of the computation C is* **location consistent** *if for all locations l there exists a topological sort T_l of C such that every read operation on location l returns the value of the last write to location l occurring in T_l.*

In previous work [27, 26], we presented ***dag consistency***, a memory model strictly weaker than location consistency. Afterwards, I showed anomalies in the definition of dag consistency, and I argued that location consistency is the weakest reasonable memory model [54]. In Chapter 5, we will use the "computation-centric" theoretical framework to understand the differences among location consistency, dag consistency, and other memory models.

We now describe the BACKER coherence algorithm from [27], in which versions of shared-memory locations can reside simultaneously in any of the processor caches and the main memory. Each processor's cache contains locations recently used by the threads that have executed on that processor, and the main memory provides default global storage for each location. In order for a thread executing on the processor to read or write a location, the location must be in the processor's cache. Each location in the cache has a ***dirty bit*** to record whether the location has been modified since it was brought into the cache.

BACKER uses three basic operations to manipulate shared-memory locations: fetch, reconcile, and flush. A ***fetch*** copies an location from the main memory to a processor cache and marks the cached location as clean. A ***reconcile*** copies a dirty location from a processor cache to the main memory and marks the cached location as clean. Finally, a ***flush*** removes a clean location from a processor cache.

The BACKER coherence algorithm operates as follows. When the user code performs a read or write operation on a location, the operation is performed directly on a cached copy of the location. If the location is not in the cache, it is fetched from the main memory before the operation is performed. If the operation is a write, the dirty bit of the location is set. To make space in the cache for a new location, a clean location can be removed by flushing it from the cache. To remove a dirty location, BACKER first reconciles and then flushes it.

Besides performing these basic operations in response to user reads and writes, BACKER performs additional reconciles and flushes to enforce location consistency. For each edge $u \to v$ in the computation dag, if nodes u and v are executed on different processors, say p and q, then BACKER causes p to reconcile all its cached locations after executing u but before enabling v, and it causes q to reconcile and flush its entire cache before executing v. Note that if q's cache is flushed for some other reason after p has reconciled its cache but before q executes v (perhaps because of another interprocessor dag edge), it need not be flushed again before executing v.

The following theorem by Luchangco states that BACKER is correct.

Theorem 24 *If the shared memory $\mathcal{M}$ of a multithreaded computation is maintained using* BACKER*, then $\mathcal{M}$ is location consistent.*

Proof: See [104]. ∎

4.3 Analysis of execution time

In this section, we bound the execution time of fully strict multithreaded computations when the parallel execution is scheduled by a work-stealing scheduler and location consistency is maintained by the BACKER algorithm, under the assumption that accesses to the main memory are random and independent. For a given fully strict multithreaded algorithm, let $T_P(Z, L)$ denote the time taken by the algorithm to solve a given problem on a parallel computer with P processors, each with an LRU (Z, L)-cache, when the execution is scheduled by the Cilk scheduler in conjunction with the BACKER coherence algorithm. In this section, we show that if accesses to main memory are random and independent, then the expected value of $T_P(Z, L)$ is $O(T_1(Z, L)/P + \mu H T_\infty)$, where $H = Z/L$ is the height of the cache, μ denotes the minimum time to transfer a cache line, and T_∞ is the critical-path length of the computation. In addition, we bound the number of cache misses. The exposition of the proofs in this section makes heavy use of results and techniques from [25, 30].

In the following analysis, we consider the fully strict multithreaded computation that results when a given fully strict multithreaded algorithm is executed to solve a given input problem. We assume that the computation is executed by a work-stealing scheduler in conjunction with the BACKER coherence algorithm on a parallel computer with P homogeneous processors. The main memory is distributed across the processors by hashing, with each processor managing a proportional share of the locations which are grouped into cache lines of size L. In addition to main memory, each processor has a cache of H lines that is maintained using the LRU replacement heuristic. We assume that a minimum of μ time steps are required to transfer a cache line. When cache lines are transferred between processors, congestion may occur at a destination processor, in which case we assume that the transfers are serviced at the destination in FIFO (first-in, first-out) order.

The work-stealing scheduler assumed in our analysis is the work-stealing scheduler from [25, 30], but with a small technical modification. Between successful steals, we wish to guarantee that a processor performs at least H line transfers (fetches or reconciles) so that it does not steal too often. Consequently, whenever a processor runs out of work, if it has not performed H line transfers since its last successful steal, the modified work-stealing scheduler performs enough additional "idle" transfers until it has transferred H lines. At that point, it can steal again. Similarly, we require that each processor perform one idle transfer after each unsuccessful steal request to ensure that steal requests do not happen too often.

Our analysis of execution time is organized as follows. First, we prove a lemma describing how the BACKER algorithm adds cache misses to a parallel execution. Then, we obtain a bound on the number of "rounds" that a parallel execution contains. Each round contains a fixed amount of scheduler overhead, so bounding the number of rounds bounds the total amount of scheduler overhead. To complete the analysis, we use an accounting argument to add up the total execution time.

Before embarking on the analysis, however, we first define some helpful terminology. A ***task*** is the fundamental building block of a computation and is either a local instruction (one that does not access shared memory) or a shared-memory operation. If a task is a local instruction or references a location in the local cache, it takes 1 step to execute. Otherwise, the task is referencing an location not in the local cache, and a line transfer occurs, taking at least μ steps to execute. A

synchronization task is a task in the dag that forces BACKER to perform a cache flush in order to maintain location consistency. Remember that for each interprocessor edge $u \rightarrow v$ in the dag, a cache flush is required by the processor executing v sometime after u executes but before v executes. A synchronization task is thus a task v having an incoming interprocessor edge $u \rightarrow v$ in the dag, where v executes on a processor that has not flushed its cache since u was executed. A ***subcomputation*** is the computation that one processor performs from the time it obtains work to the time it goes idle or enables a synchronization task. We distinguish two kinds of subcomputations: ***primary*** subcomputations start when a processor obtains work from a random steal request, and ***secondary*** subcomputations start when a processor starts executing from a synchronization task. We distinguish three kinds of line transfers. An ***intrinsic*** transfer is a transfer that would occur during a 1-processor depth-first execution of the computation. The remaining ***extrinsic*** line transfers are divided into two types. A ***primary*** transfer is any extrinsic transfer that occurs during a primary subcomputation. Likewise, a ***secondary*** transfer is any extrinsic transfer that occurs during a secondary subcomputation. We use these terms to refer to cache misses as well.

Lemma 25 *Each primary transfer during an execution can be associated with a currently running primary subcomputation such that each primary subcomputation has at most $3H$ associated primary transfers. Similarly, each secondary transfer during an execution can be associated with a currently running secondary subcomputation such that each secondary subcomputation has at most $3H$ associated secondary transfers.*

Proof: For this proof, we use a fact shown in [27] that executing a subcomputation starting with an arbitrary cache can only incur H more cache misses than the same block of code incurred in the serial execution. This fact follows from the observation that a subcomputation is executed in the same depth-first order as it would have been executed in the serial execution, and the fact that the cache replacement strategy is LRU.

We associate each primary transfer with a running primary subcomputation as follows. During a steal, we associate the (at most) H reconciles done by the victim with the stealing subcomputation. In addition, the stolen subcomputation has at most H extrinsic cache misses, because the stolen subcomputation is executed in the same order as the subcomputation executes in the serial order. At the end of the subcomputation, at most H lines need be reconciled, and these reconciles may be extrinsic transfers. In total, at most $3H$ primary transfers are associated with any primary subcomputation.

A similar argument holds for secondary transfers. Each secondary subcomputation must perform at most H reconciles to flush the cache at the start of the subcomputation. The subcomputation then has at most H extrinsic cache misses during its execution, because it executes in the same order as it executes in the serial order. Finally, at most H lines need to be reconciled at the end of the subcomputation. ∎

We now bound the amount of scheduler overhead by counting the number of rounds in an execution.

Lemma 26 *If each line transfer (fetch or reconcile) in the execution is serviced by a processor chosen independently at random, and each processor queues its transfer requests in FIFO order, then, for any $\epsilon > 0$, with probability at least $1 - \epsilon$, the total number of steal requests and primary transfers is at most $O(HPT_\infty + HP \lg(1/\epsilon))$.*

Proof: To begin, we shall assume that each access to the main memory takes one step regardless of the congestion. We shall describe how to handle congestion at the end of the proof.

First, we wish to bound the overhead of scheduling, that is, the additional work that the one-processor execution would not need to perform. We define an ***event*** as either the sending of a steal request or the sending of a primary line-transfer request. In order to bound the number of events, we divide the execution into rounds. Round 1 starts at time step 1 and ends at the first time step at which at least $27HP$ events have occurred. Round 2 starts one time step after round 1 completes and ends when it contains at least $27HP$ events, and so on. We shall show that with probability at least $1 - \epsilon$, an execution contains only $O(T_\infty + \lg(1/\epsilon))$ rounds.

To bound the number of rounds, we shall use a delay-sequence argument. We define a modified dag $\mathcal{G}'$ exactly as in [30]. (The dag $\mathcal{G}'$ is for the purposes of analysis only and has no effect on the computation.) The critical-path length of $\mathcal{G}'$ is at most $2T_\infty$. We define a task with no unexecuted predecessors in $\mathcal{G}'$ to be ***critical***, and it is by construction one of the first two tasks to be stolen from the processor on which it resides. Given a task that is critical at the beginning of a round, we wish to show that it is executed by the start of the next round with constant probability. This fact will enable us to show that progress is likely to be made on any path of $\mathcal{G}'$ in each round.

We now show that at least $4P$ steal requests are initiated during the first $22HP$ events of a round. If at least $4P$ of the $22HP$ events are steal requests, then we are done. If not, then there are at least $18HP$ primary transfers. By Lemma 25, we know that at most $3HP$ of these transfers are associated with subcomputations running at the start of the round, leaving $15HP$ for steals that start in this round. Since at most $3H$ primary transfers can be associated with any steal, at least $5P$ steals must have occurred. At most P of these steals were requested in previous rounds, so there must be at least $4P$ steal requests in this round.

We now argue that any task that is critical at the beginning of a round has a probability of at least $1/2$ of being executed by the end of the round. Since there are at least $4P$ steal requests during the first $22HP$ events, the probability is at least $1/2$ that any task that is critical at the beginning of a round is the target of a steal request [30, Lemma 10], if it is not executed locally by the processor on which it resides. Any task takes at most $3\mu H + 1 \leq 4\mu H$ time to execute, since we are ignoring the effects of congestion for the moment. Since the last $4HP$ events of a round take at least $4\mu H$ time to execute, if a task is stolen in the first part of the round, it is done by the end of the round.

We want to show that with probability at least $1 - \epsilon$, the total number of rounds is $O(T_\infty + \lg(1/\epsilon))$. Consider a possible delay sequence. Recall from [30] that a delay sequence of size R is a maximal path U in the augmented dag $\mathcal{G}'$ of length at most $2T_\infty$, along with a partition Π of R which represents the number of rounds during which each task of the path in $\mathcal{G}'$ is critical. We now show that the probability of a large delay sequence is tiny.

Whenever a task on the path U is critical at the beginning of a round, it has a probability of at least $1/2$ of being executed during the round, because it is likely to be the target of one of the $4P$ steals in the first part of the round. Furthermore, this probability is independent of the success of critical tasks in previous rounds, because victims are chosen independently at random. Thus, the probability is at most $(1/2)^{R-2T_\infty}$ that a particular delay sequence with size $R > 2T_\infty$ actually occurs in an execution. There are at most $2^{2T_\infty}\binom{R+2T_\infty}{2T_\infty}$ delay sequences of size R. Thus, the probability that any delay sequence of size R occurs is at most

$$\begin{aligned}
&2^{2T_\infty}\binom{R+2T_\infty}{2T_\infty}\left(\frac{1}{2}\right)^{R-2T_\infty} \\
&\quad \le\ 2^{2T_\infty}\left(\frac{e(R+2T_\infty)}{2T_\infty}\right)^{2T_\infty}\left(\frac{1}{2}\right)^{R-2T_\infty} \\
&\quad \le\ \left(\frac{4e(R+2T_\infty)}{2T_\infty}\right)^{2T_\infty}\left(\frac{1}{2}\right)^{R},
\end{aligned}$$

which can be made less than ϵ by choosing $R = 14T_\infty + \lg(1/\epsilon)$. Therefore, there are at most $O(T_\infty + \lg(1/\epsilon))$ rounds with probability at least $1 - \epsilon$. In each round, there are at most $28HP$ events, so there are at most $O(HPT_\infty + HP\lg(1/\epsilon))$ steal requests and primary transfers in total.

Now, let us consider what happens when congestion occurs at the main memory. We still have at most $3H$ transfers per task, but these transfers may take more than $3\mu H$ time to complete because of congestion. We define the following indicator random variables to keep track of the congestion. Let x_{uip} be the indicator random variable that tells whether task u's ith transfer request is delayed by a transfer request from processor p. The probability is at most $1/P$ that one of these indicator variables is 1. Furthermore, we shall argue that they are nonpositively correlated, that is, $\Pr\left\{x_{uip} = 1 \,\middle|\, \bigwedge_{u'i'p'} x_{u'i'p'} = 1\right\} \le 1/P$, as long as none of the (u', i') requests execute at the same time as the (u, i) request. That they are nonpositively correlated follows from an examination of the queuing behavior at the main memory. If a request (u', i') is delayed by a request from processor p' (that is, $x_{u'i'p'} = 1$), then once the (u', i') request has been serviced, processor p''s request has also been serviced, because we have FIFO queuing of transfer requests. Consequently, p''s next request, if any, goes to a new, random processor when the (u, i) request occurs. Thus, a long delay for request (u', i') cannot adversely affect the delay for request (u, i). Finally, we also have $\Pr\left\{x_{uip} = 1 \,\middle|\, \bigwedge_{p' \ne p} x_{uip'} = 1\right\} \le 1/P$, because the requests from the other processors besides p are distributed at random.

The execution time X of the transfer requests for a path U in $\mathcal{G}'$ can be written as $X \leq \sum_{u \in U}(5\mu H + \mu \sum_{ip} x_{uip})$. Rearranging, we have $X \leq 10\mu H T_\infty + \mu \sum_{uip} x_{uip}$, because U has length at most $2T_\infty$. This sum is just the sum of $10HPT_\infty$ indicator random variables, each with expectation at most $1/P$. Since the tasks u in U do not execute concurrently, the x_{uip} are non-positively correlated, and thus, their sum can be bounded using combinatorial techniques. The sum is greater than z only if some z-size subset of these $10HPT_\infty$ variables are all 1, which happens with probability:

$$\begin{aligned}
\Pr\left\{\sum_{uip} x_{uip} \geq z\right\} &\leq \binom{10HPT_\infty}{z}\left(\frac{1}{P}\right)^z \\
&\leq \left(\frac{10eHPT_\infty}{z}\right)^z \left(\frac{1}{P}\right)^z \\
&\leq \left(\frac{10eHT_\infty}{z}\right)^z .
\end{aligned}$$

This probability can be made less than $(1/2)^z$ by choosing $z \geq 20eHT_\infty$. Therefore, we have $X > (10 + 20e)\mu H T_\infty$ with probability at most $(1/2)^{X - 10\mu H T_\infty}$. Since there are at most $2T_\infty$ tasks on the critical path, at most $2T_\infty + X/\mu H$ rounds can be overlapped by the long execution of line transfers of these critical tasks. Therefore, the probability of a delay sequence of size R is at most $(1/2)^{R - O(T_\infty)}$. Consequently, we can apply the same argument as for unit-cost transfers, with slightly different constants, to show that with probability at least $1 - \epsilon$, there are $O(T_\infty + \lg(1/\epsilon))$ rounds, and hence $O(HPT_\infty + HP\lg(1/\epsilon))$ events, during the execution. ■

We now bound the running time of a computation.

Theorem 20 *Consider any fully strict multithreaded computation executed on P processors, each with an LRU (Z, L)-cache of height H, using the Cilk work-stealing scheduler in conjunction with the* BACKER *coherence algorithm. Let μ be the service time for a cache miss that encounters no congestion, and assume that accesses to the main memory are random and independent. Suppose the computation has T_1 computational work, $Q(Z, L)$ serial cache misses, $T_1(Z, L) = T_1 + \mu Q(Z, L)$ total work, and T_∞ critical-path length. Then for any $\epsilon > 0$, the execution time is $O(T_1(Z, L)/P + \mu H T_\infty + \mu P \lg P + \mu H \lg(1/\epsilon))$ with probability at least $1 - \epsilon$. Moreover, the expected execution time is $O(T_1(Z, L)/P + \mu H T_\infty)$.*

Proof: As in [30], we shall use an accounting argument to bound the running time. During the execution, at each time step, each processor puts a piece of silver into one of 5 buckets according to its activity at that time step. Specifically, a processor puts a piece of silver in the bucket labeled:

- **WORK**, if the processor executes a task;
- **STEAL**, if the processor sends a steal request;

- **STEALWAIT**, if the processor waits for a response to a steal request;
- **XFER**, if the processor sends a line-transfer request; and
- **XFERWAIT**, if the processor waits for a line transfer to complete.

When the execution completes, we add up the pieces of silver in each bucket and divide by P to get the running time.

We now bound the amount of money in each of the buckets at the end of the computation by using the fact, from Lemma 26, that with probability at least $1 - \epsilon'$, there are $O(HPT_\infty + HP\lg(1/\epsilon'))$ events:

WORK. The WORK bucket contains exactly T_1 pieces of silver, because there are exactly T_1 tasks in the computation.

STEAL. We know that there are $O(HPT_\infty + HP\lg(1/\epsilon'))$ steal requests, so there are $O(HPT_\infty + HP\lg(1/\epsilon'))$ pieces of silver in the STEAL bucket.

STEALWAIT. We use the analysis of the ***recycling game*** ([30, Lemma 5]) to bound the number of pieces of silver in the STEALWAIT bucket. The recycling game says that if N requests are distributed randomly to P processors for service, with at most P requests outstanding simultaneously, the total time waiting for the requests to complete is $O(N + P\lg P + P\lg(1/\epsilon'))$ with probability at least $1 - \epsilon'$. Since steal requests obey the assumptions of the recycling game, if there are $O(HPT_\infty + HP\lg(1/\epsilon'))$ steals, then the total time waiting for steal requests is $O(HPT_\infty + P\lg P + HP\lg(1/\epsilon'))$ with probability at least $1 - \epsilon'$. We must add to this total an extra $O(\mu HPT_\infty + \mu HP\lg(1/\epsilon'))$ pieces of silver because the processors initiating a successful steal must also wait for the cache of the victim to be reconciled, and we know that there are $O(HPT_\infty + HP\lg(1/\epsilon'))$ such reconciles. Finally, we must add $O(\mu HPT_\infty + \mu HP\lg(1/\epsilon))$ pieces of silver because each steal request might also have up to μ idle steps associated with it. Thus, with probability at least $1 - \epsilon'$, we have a total of $O(\mu HPT_\infty + P\lg P + \mu HP\lg(1/\epsilon'))$ pieces of silver in the STEALWAIT bucket.

XFER. We know that there are $O(Q(Z, L) + HPT_\infty + HP\lg(1/\epsilon'))$ transfers during the execution: a fetch and a reconcile for each intrinsic miss, $O(HPT_\infty + HP\lg(1/\epsilon'))$ primary transfers from Lemma 26, and $O(HPT_\infty + HP\lg(1/\epsilon'))$ secondary transfers. We have this bound on secondary transfers, because each secondary subcomputation can be paired with a unique primary subcomputation. We construct this pairing as follows. For each synchronization task v, we examine each interprocessor edge entering v. Each of these edges corresponds to some child of v's thread in the spawn tree, because the computation is fully strict. At least one of these children (call it w) is not finished executing at the time of the last cache flush by v's processor, since v is a synchronization task. We now show that there must be a random steal of v's thread just after w is spawned. If not, then w is completed before v's thread continues executing after the spawn. There must be a random steal somewhere between when w is spawned and when v is executed, however, because v and w execute on different processors. On the last such random steal, the processor executing v must flush its cache, but this cannot happen because w is still executing when the last flush of the cache occurs. Thus, there must be a random steal just after w is spawned. We pair the secondary subcomputation that starts at task v with the primary subcomputation that starts with the random steal after w is spawned. By construction, each primary subcomputation has at most one secondary subcomputation paired with it, and since each primary subcomputation does at least H extrinsic transfers and each secondary subcomputation does at most $3H$ extrinsic transfers, there are at most $O(HPT_\infty + HP \lg(1/\epsilon'))$ secondary transfers. Since each transfer takes μ time, the number of pieces of silver in the XFER bucket is $O(\mu Q(Z, L) + \mu HPT_\infty + \mu HP \lg(1/\epsilon'))$.

XFERWAIT. To bound the pieces of silver in the XFERWAIT bucket, we use the recycling game as we did for the STEALWAIT bucket. The recycling game shows that there are $O(\mu Q(Z, L) + \mu HPT_\infty + \mu P \lg P + \mu HP \lg(1/\epsilon'))$ pieces of silver in the XFERWAIT bucket with probability at least $1 - \epsilon'$.

With probability at least $1 - 3\epsilon'$, the sum of all the pieces of silver in all the buckets is $T_1 + O(\mu Q(Z, L) + \mu HPT_\infty + \mu P \lg P + \mu HP \lg(1/\epsilon'))$. Dividing by P, we obtain a running time of $T_P \le O((T_1 + \mu Q(Z, L))/P + \mu HT_\infty + \mu \lg P + \mu H \lg(1/\epsilon'))$ with probability at least $1 - 3\epsilon'$. Using the identity $T_1(Z, L) = T_1 + \mu Q(Z, L)$ and substituting $\epsilon = 3\epsilon'$ yields the desired high-probability bound. The expected bound follows similarly. ■

To conclude this section, we now bound the number of cache misses.

Corollary 21 *Consider any fully strict multithreaded computation executed on P processors, each with an LRU (Z, L)-cache of height H, using the Cilk work-stealing scheduler in conjunction with the* BACKER *coherence algorithm. Assume that accesses to the main memory are random and independent. Suppose the computation has $Q(Z, L)$ serial cache misses and T_∞ critical-path length. Then for any $\epsilon > 0$, the number of cache misses is at most $Q(Z, L) + O(HPT_\infty + HP \lg(1/\epsilon))$ with probability at least $1 - \epsilon$. Moreover, the expected number of cache misses is at most $Q(Z, L) + O(HPT_\infty)$.*

Proof: In the parallel execution, we have one miss for each intrinsic miss, plus an extra $O(HPT_\infty + HP \lg(1/\epsilon))$ primary and secondary misses. The expected bound follows similarly. ■

4.4 Analysis of space utilization

This section provides upper bounds on the memory requirements of "regular" divide-and-conquer multithreaded algorithms when the parallel execution is scheduled by a "busy-leaves" scheduler, such as the work-stealing scheduler used by Cilk. A ***busy-leaves*** scheduler is a scheduler with the property that at all times during the execution, if a thread has no living children, then that thread has a processor working on it. The work-stealing scheduler is a busy-leaves scheduler [25, 30]. In a ***regular divide-and-conquer multithreaded algorithm***, each thread, when spawned to solve a problem of size n, operates as follows. If n is larger than some given constant, the thread divides the problem into a subproblems, each of size n/b for some constants $a \geq 1$ and $b > 1$, and then it recursively spawns child threads to solve each subproblem. When all a of the children have completed, the thread merges their results, and then returns. In the base case, when n is smaller than the specified constant, the thread directly solves the problem, and then returns. We shall proceed through a series of lemmas that provide an exact characterization of the space used by "simple" multithreaded algorithms when executed by a busy-leaves scheduler. A ***simple multithreaded algorithm*** is a fully strict multithreaded algorithm in which each thread's control consists of allocating memory, spawning children, waiting for the children to complete, deallocating memory, and returning, in that order. We shall then specialize this characterization to provide space bounds for regular divide-and-conquer algorithms.

Previous work [25, 30] has shown that a busy-leaves scheduler can efficiently execute a fully strict multithreaded algorithm on P processors using no more space than P times the space required to execute the algorithm on a single processor. Specifically, for a given fully strict multithreaded algorithm, if S_1 denotes the space used by the algorithm to solve a given problem with the standard, depth-first, serial execution order, then for any number P of processors, a busy leaves scheduler uses at most PS_1 space. The basic idea in the proof of this bound is that a busy-leaves scheduler never allows more than P leaves in the spawn tree of the resulting computation to be living at one time. If we look at any path in the spawn tree from the root to a leaf and add up all the space allocated on that path, the largest such value we can obtain is S_1. The bound then follows, because each of the at most P leaves living at any time is responsible for at most S_1 space, for a total of PS_1 space. For many algorithms, however, the bound PS_1 is an overestimate of the true space, because space near the root of the spawn tree may be counted multiple times. In this section, we tighten this bound for the case of regular divide-and-conquer algorithms. We start by considering the more general case of simple multithreaded algorithms.

We first introduce some terminology. Consider any simple multithreaded algorithm and input problem, and let $\mathcal{T}$ be the spawn tree of the simple multithreaded computation that results when the given algorithm is executed to solve the given problem. Let Λ be any nonempty set of the leaves of $\mathcal{T}$. A node (thread) $u \in \mathcal{T}$ is ***covered*** by Λ if u lies on the path from some leaf in Λ to the root of $\mathcal{T}$. The ***cover*** of Λ, denoted $\mathcal{C}(\Lambda)$, is the set of nodes covered by Λ. Since all nodes on the path from any node in $\mathcal{C}(\Lambda)$ to the root are covered, it follows that $\mathcal{C}(\Lambda)$ is connected and forms a subtree of $\mathcal{T}$. If each node u allocates $f(u)$ memory, then the space used by Λ is defined as

$$S(\Lambda) = \sum_{u \in \mathcal{C}(\Lambda)} f(u) .$$

The following lemma shows how the notion of a cover can be used to characterize the space required by a simple multithreaded algorithm when executed by a busy leaves scheduler.

Lemma 27 *Let $\mathcal{T}$ be the spawn tree of a simple multithreaded computation, and let $f(u)$ denote the memory allocated by node $u \in \mathcal{T}$. For any number P of processors, if the computation is executed using a busy-leaves scheduler, then the total amount of allocated memory at any time during the execution is at most S^*, which we define by the identity*

$$S^* = \max_{|\Lambda| \leq P} S(\Lambda) ,$$

with the maximum taken over all sets Λ of leaves of $\mathcal{T}$ of size at most P.

Proof: Consider any given time during the execution, and let Λ denote the set of leaves living at that time, which by the busy-leaves property has cardinality at most P. The total amount of allocated memory is the sum of the memory allocated by the leaves in Λ plus the memory allocated by all their ancestors. Since both leaves and ancestors belong to $\mathcal{C}(\Lambda)$ and $|\Lambda| \leq P$ holds, the lemma follows. ■

The next few definitions will help us characterize the structure of $\mathcal{C}(\Lambda)$ when Λ maximizes the space used. Let $\mathcal{T}$ be the spawn tree of a simple multithreaded computation, and let $f(u)$ denote the memory allocated by node $u \in \mathcal{T}$, where we shall henceforth make the technical assumption that $f(u) = 0$ holds if u is a leaf and $f(u) > 0$ holds if u is an internal node. When necessary, we can extend the spawn tree with a new level of leaves in order to meet this technical assumption. Define the ***serial-space function*** $S(u)$ inductively on the nodes of $\mathcal{T}$ as follows:

$$S(u) = \begin{cases} 0 & \text{if } u \text{ is a leaf;} \\ f(u) + \max\{S(v) : v \text{ is a child of } u\} & \text{otherwise} \end{cases} .$$

The serial-space function assumes a strictly increasing sequence of values on the path from any leaf to the root. Moreover, for each node $u \in \mathcal{T}$, there exists a leaf such that if Π is the unique simple path from u to that leaf, then we have $S(u) = \sum_{v \in \Pi} f(v)$. We shall denote that leaf (or an arbitrary such leaf, if more than one exists) by $\lambda(u)$. The ***u-induced dominator*** of a set Λ of leaves of $\mathcal{T}$ is

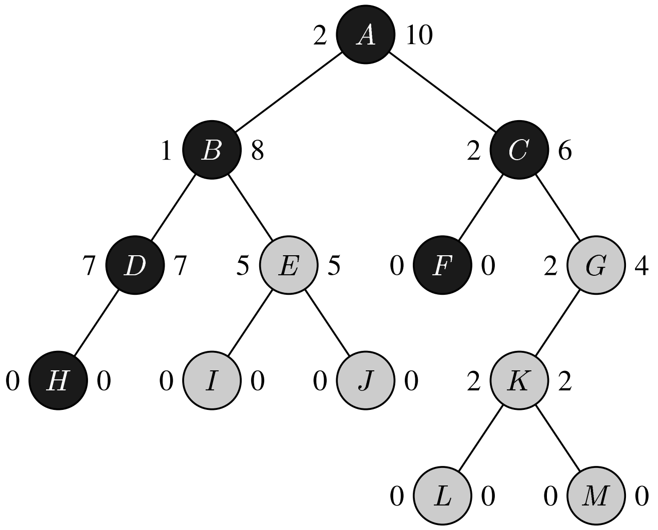


Figure 4-2: An illustration of the definition of a dominator set. For the tree shown, let f be given by the labels at the left of the nodes, and let $\Lambda = \{F, H\}$. Then, the serial space S is given by the labels at the right of the nodes, $\mathcal{C}(\Lambda) = \{A, B, C, D, F, H\}$ (the dark nodes), and $\mathcal{D}(\Lambda, G) = \{C, D\}$. The space required by Λ is $S(\Lambda) = 12$.

defined by

$$\mathcal{D}(\Lambda, u) = \{v \in \mathcal{T} : \exists w \in \mathcal{C}(\Lambda) \text{ such that } w \text{ is a child of } v \text{ and } S(w) < S(u) \leq S(v)\} .$$

The next lemma shows that every induced dominator of Λ is indeed a "dominator" of Λ.

Lemma 28 *Let $\mathcal{T}$ be the spawn tree of a simple multithreaded computation encompassing more than one node, and let Λ be a nonempty set of leaves of $\mathcal{T}$. Then, for any internal node $u \in \mathcal{T}$, removal of $\mathcal{D}(\Lambda, u)$ from $\mathcal{T}$ disconnects each leaf in Λ from the root of $\mathcal{T}$.*

Proof: Let r be the root of $\mathcal{T}$, and consider the path Π from any leaf $l \in \Lambda$ to r. We shall show that some node on the path belongs to $\mathcal{D}(\Lambda, u)$. Since u is not a leaf and S is strictly increasing on the nodes of the path Π, we must have $0 = S(l) < S(u) \leq S(r)$. Let w be the node lying on Π that maximizes $S(w)$ such that $S(w) < S(u)$ holds, and let v be its parent. We have $S(w) < S(u) \leq S(v)$ and $w \in \mathcal{C}(\Lambda)$, because all nodes lying on Π belong to $\mathcal{C}(\Lambda)$, which implies that $v \in \mathcal{D}(\Lambda, u)$ holds. ■

The next lemma shows that whenever we have a set Λ of leaves that maximizes space, every internal node u not covered by Λ induces a dominator that is at least as large as Λ.

Lemma 29 *Let $\mathcal{T}$ be the spawn tree of a simple multithreaded computation encompassing more than one node, and for any integer $P \geq 1$, let Λ be a set of leaves such that $S(\Lambda) = S^*$ holds. Then, for all internal nodes $u \notin \mathcal{C}(\Lambda)$, we have $|\mathcal{D}(\Lambda, u)| \geq |\Lambda|$.*

Proof: Suppose, for the purpose of contradiction, that $|\mathcal{D}(\Lambda, u)| < |\Lambda|$ holds. Lemma 28 implies that each leaf in Λ is a descendant of some node in $\mathcal{D}(\Lambda, u)$. Consequently, by the pigeonhole principle, there must exist a node $v \in \mathcal{D}(\Lambda, u)$ that is ancestor of at least two leaves in Λ. By the definition of induced dominator, a child $w \in \mathcal{C}(\Lambda)$ of v must exist such that $S(w) < S(u)$ holds.

We shall now show that a new set Λ' of leaves can be constructed such that we have $S(\Lambda') > S(\Lambda)$, thus contradicting the assumption that the function S achieves its maximum value on Λ. Since w is covered by Λ, the subtree rooted at w must contain a leaf $l \in \Lambda$. Define $\Lambda' = \Lambda - \{l\} \cup \{\lambda(u)\}$. Adding $\lambda(u)$ to Λ causes the value of $S(\Lambda)$ to increase by at least $S(u)$, and the removal of l causes the path from l to some descendant of w (possibly w itself) to be removed, thus decreasing the value of $S(\Lambda)$ by at most $S(w)$. Therefore, we have $S(\Lambda') \geq S(\Lambda) - S(w) + S(u) > S(\Lambda)$, since $S(w) < S(u)$ holds. ■

We now restrict our attention to regular divide-and-conquer multithreaded algorithms. In a regular divide-and-conquer multithreaded algorithm, each thread, when spawned to solve a problem of size n, allocates an amount of space $s(n)$ for some function s of n. The following lemma characterizes the structure of the worst-case space usage for this class of algorithms.

Lemma 30 *Let $\mathcal{T}$ be the spawn tree of a regular divide-and-conquer multithreaded algorithm encompassing more than one node, and for any integer $P \geq 1$, let Λ be a set of leaves such that $S(\Lambda) = S^*$ holds. Then, $\mathcal{C}(\Lambda)$ contains every node at every level of the tree with P or fewer nodes.*

Proof: If $\mathcal{T}$ has fewer than P leaves, then Λ consists of all the leaves of $\mathcal{T}$ and the lemma follows trivially. Thus, we assume that $\mathcal{T}$ has at least P leaves, and we have $|\Lambda| = P$.

Suppose now, for the sake of contradiction, that there is a node u at a level of the tree with P or fewer nodes such that $u \notin \mathcal{C}(\Lambda)$ holds. Since all nodes at the same level of the spawn tree allocate the same amount of space, the set $\mathcal{D}(\Lambda, u)$ consists of all covered nodes at the same level as u, all of which have the same serial space $S(u)$. Lemma 29 then says that there are at least P nodes at the same level as u that are covered by Λ. This fact contradicts our assumption that the tree has P or fewer nodes at the same level as u. ■

We are now ready to prove Theorem 22 from Section 4.1, which bounds the worst-case space used by a regular divide-and-conquer multithreaded algorithm when it is scheduled using a busy-leaves scheduler.

Theorem 22 *Consider any regular divide-and-conquer multithreaded algorithm executed on P processors using a busy-leaves scheduler. Suppose that each thread, when spawned to solve a* *problem of size n, allocates $s(n)$ space, and if n is larger than some constant, then the thread divides the problem into a subproblems each of size n/b for some constants $a \geq 1$ and $b > 1$. Then, the total amount $S_P(n)$ of space taken by the algorithm in the worst case when solving a problem of size n can be determined as follows:*[3]

1. *If $s(n) = \Theta(\lg^k n)$ for some constant $k \geq 0$, then $S_P(n) = \Theta(P \lg^{k+1}(n/P))$.*

2. *If $s(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $S_P(n) = \Theta(Ps(n/P^{1/\log_b a}))$, if, in addition, $s(n)$ satisfies the regularity condition $\gamma_1 s(n/b) \leq s(n) \leq a\gamma_2 s(n/b)$ for some constants $\gamma_1 > 1$ and $\gamma_2 < 1$.*

3. *If $s(n) = \Theta(n^{\log_b a})$, then $S_P(n) = \Theta(s(n) \lg P)$.*

4. *If $s(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, then $S_P(n) = \Theta(s(n))$, if, in addition, $s(n)$ satisfies the regularity condition that $s(n) \geq a\gamma s(n/b)$ for some constant $\gamma > 1$.*

Proof: Consider the spawn tree $\mathcal{T}$ of the multithreaded computation that results when the algorithm is used to solve a given input problem of size n. The spawn tree $\mathcal{T}$ is a perfectly balanced a-ary tree. A node u at level k in the tree allocates space $f(u) = s(n/b^k)$. From Lemma 27 we know that the maximum space usage is bounded by S^*, which we defined as the maximum value of the space function $S(\Lambda)$ over all sets Λ of leaves of the spawn tree having size at most P.

In order to bound the maximum value of $S(\Lambda)$, we shall appeal to Lemma 30 which characterizes the set Λ at which this maximum occurs. Lemma 30 states that for this set Λ, the set $\mathcal{C}(\Lambda)$ contains every node in the first $\lfloor \log_a P \rfloor$ levels of the spawn tree. Thus, we have

$$S_P(n) \leq \sum_{i=0}^{\lfloor \log_a P \rfloor - 1} a^i s(n/b^i) + \Theta(PS_1(n/P^{1/\log_b a})) \ . \tag{4.1}$$

To determine which term in Equation (4.1) dominates, we must evaluate $S_1(n)$, which satisfies the recurrence

$$S_1(n) = S_1(n/b) + s(n) \ ,$$

because with serial execution the depth-first discipline allows each of the a subproblems to reuse the same space. The solution to this recurrence [42, Section 4.4] is

- $S_1(n) = \Theta(\lg^{k+1} n)$, if $s(n) = \Theta(\lg^k n)$ for some constant $k \geq 0$, and
- $S_1(n) = \Theta(s(n))$, if $s(n) = \Omega(n^\epsilon)$ for some constant $\epsilon > 0$ and in addition satisfies the regularity condition that $s(n) \geq \gamma s(n/b)$ for some constant $\gamma > 1$.

[3] *Other cases exist besides those given here.*

The theorem follows by evaluating Equation (4.1) for each of the cases. We only sketch the essential ideas in the algebraic manipulations. For Cases 1 and 2, the serial space dominates, and we simply substitute appropriate values for the serial space. In Cases 3 and 4, the space at the top of the spawn tree dominates. In Case 3, the total space at each level of the spawn tree is the same. In Case 4, the space at each level of the spawn tree decreases geometrically, and thus, the space allocated by the root dominates the entire tree. ∎

4.5 Related work

Like Cilk's location consistency, most distributed shared memories (DSM's) employ a relaxed consistency model in order to realize performance gains, but unlike location consistency, most distributed shared memories take a low-level view of parallel programs and cannot give analytical performance bounds. Relaxed shared-memory consistency models are motivated by the fact that sequential consistency [96] and various forms of processor consistency [70] are too expensive to implement in a distributed setting. (Even modern "symmetric multiprocessors" do not typically implement sequential consistency.) Relaxed models, such as location consistency [60] and various forms of release consistency [3, 47, 64], ensure consistency (to varying degrees) only when explicit synchronization operations occur, such as the acquisition or release of a lock. Causal memory [7] ensures consistency only to the extent that if a process A reads a value written by another process B, then all subsequent operations by A must appear to occur after the write by B. Most DSM's implement one of these relaxed consistency models [33, 87, 89, 130], though some implement a fixed collection of consistency models [20], while others merely implement a collection of mechanisms on top of which users write their own DSM consistency policies [97, 128]. All of these consistency models and the DSM's that implement these models take a low-level view of a parallel program as a collection of cooperating processes.

In contrast, location consistency takes the high-level view of a parallel program as a dag, and this dag exactly defines the memory consistency required by the program. (This perspective is elaborated in Chapter 5.) Like some of these other DSM's, location consistency allows synchronization to affect only the synchronizing processors and does not require a global broadcast to update or invalidate data. Unlike these other DSM's, however, location consistency requires no extra bookkeeping overhead to keep track of which processors might be involved in a synchronization operation, because this information is encoded explicitly in the dag. By leveraging this high-level knowledge, the BACKER algorithm in conjunction with the work-stealing scheduler is able to execute multithreaded algorithms with the performance bounds shown here. The BLAZE parallel language [109] and the Myrias parallel computer [19] define a high-level relaxed consistency model much like location consistency, but we do not know of any efficient implementation of either of these systems. After an extensive literature search, we are aware of no other distributed shared memory with analytical performance bounds for any nontrivial algorithms.

4.6 Conclusion

Location consistency gives a framework that unifies the performance guarantees of Cilk and cache-oblivious algorithms. Using the BACKER coherence algorithm and the analytical bounds of Theorem 20, we can design portable algorithms that cope with both parallelism and memory hierarchies efficiently.

For portability across both parallelism and memory hierarchies, the central problem is the identification of the "right" memory model and of an appropriate coherence protocol, but many current shared-memory designs are inadequate in this respect. For example, I recently helped Don Dailey to tune the Cilkchess chess program for the forthcoming world championship. Cilkchess will be running on a 256-processor SGI Origin 2000, thanks to the generosity of NASA and SGI. This is an experimental machine installed at NASA Ames Research Center, and it is not available commercially. During the development of Cilkchess, the performance of the program suddenly dropped by a factor of about 100 after introducing a minor change. The problem turned out to be caused by a shared memory location: Every processor was writing to this location at the same time. More annoyingly, we observed similar cases of performance degradation because of *false* sharing, in which processors were writing in parallel to different locations that happened to be allocated on the same cache line. It is very hard to program for portability on such a system. For Cilkchess, however, portability is fundamental, because the program is developed on many platforms ranging from Linux laptops to supercomputers like the Origin 2000. A programming system built on top of Cilk and BACKER would have guaranteed performance and no such bad surprises.

I do not expect the results in this chapter to be the ultimate technique for portability across parallelism and memory hierarchies. BACKER is a simple protocol that might perform unnecessary communication; it is likely that more efficient protocols can be devised for which we can still preserve the performance guarantees. Location consistency is too weak for certain applications, although it is sufficient in surprisingly many cases. For these applications, Cilk-5 provides a stronger memory model through mutual-exclusion locks, but these locks are a sort of afterthought and they break all performance guarantees.

Our work to date leaves open several analytical questions regarding the performance of multithreaded algorithms that use location consistent shared memory. We would like to improve the analysis of execution time to directly account for the cost of cache misses when lines are hashed to main memory instead of assuming that accesses to main memory "appear" to be independent and random as assumed here.

Chapter 5

A theory of memory models

In Chapter 4, we identified location consistency as the memory model that allowed us to preserve Cilk's performance guarantees in the presence of hierarchical memory. This chapter elaborates on the idea of defining memory models based only on ***computations*** such as the multithreaded computations generated by Cilk. This idea was implicit in Chapter 4, where it was just ancillary to the performance analysis, and now now we develop its implications.

A memory model specifies the values that may be returned by the memory of a computer system in response to instructions issued by a program. In this chapter, we develop a ***computation-centric theory*** of memory models in which we can reason about memory models abstractly. We define formally what a memory model is, and we investigate the implications of ***constructibility***, an abstract property which is necessary for a model to be maintainable exactly by an online algorithm. The computation-centric theory is based on the two concepts of a ***computation*** and an ***observer function***.

The computation-centric theory is not directly concerned with the topic of this dissertation, which is portable high performance. Historically, however, this theory played a crucial role in convincing me that location consistency is the "right" memory model of Cilk [54], as opposed to the "dag consistency" memory model that we used in [27, 26]. I include the computation-centric theory in this dissertation because it introduces concepts, such as constructibility, that I think will be important to other researchers who want to improve upon location consistency and BACKER.

Most existing memory models [47, 3, 70, 64, 90, 20, 84] are expressed in terms of *processors* acting on *memory*. We call these memory models ***processor-centric***; the memory model specifies what happens when a processor performs some action on memory. In contrast, the philosophy of the computation-centric theory is to separate the logical dependencies among instructions (the computation) from the way instructions are mapped to processors (the schedule). For example, in a multithreaded program, the programmer specifies several execution threads and certain dependen-

This chapter represents joint work with Victor Luchangco. A preliminary version appears in [57].

cies among the threads, and expects the behavior of the program to be specified independently of which processor happens to execute a particular thread. Computation-centric memory models focus on the computation alone, and not on the schedule. While the processor-centric description has the advantage of modeling real hardware closely, our approach allows us to define formal properties of memory models that are independent of any implementation.

A ***computation*** is an abstraction of a parallel instruction stream. The computation specifies machine instructions and dependencies among them. A computation does not model a parallel program, but rather the way a program unfolds in a particular execution. (A program may unfold in different ways because of input values and nondeterministic or random choices.) We model the result of this unfolding process by a directed acyclic graph whose nodes represent instances of instructions in the execution. For example, a computation could be generated using a multithreaded language with fork/join parallelism (such as Cilk). Computations are by no means limited to modeling multithreaded programs, however. In this chapter, we assume that the computation is given, and defer the important problem of determining which computations a given program generates. We can view computations as providing a means for *post mortem* analysis, to verify whether a system meets a specification by checking its behavior after it has finished executing.

To specify memory semantics, we use the notion of an ***observer function*** for a computation. Informally, for each node of the computation (i.e., an instance of an instruction) that reads a value from the memory, the observer function specifies the node that wrote the value that the read operation receives. Computation-centric memory models are defined by specifying a set of valid observer functions for each computation. A memory implements a memory model if, for every computation, it always generates an observer function belonging to the model.

Within the computation-centric theory, we define a property we call ***constructibility***. Informally, a nonconstructible memory model cannot be implemented exactly by an online algorithm; any online implementation of a nonconstructible memory must maintain a strictly stronger constructible model. We find constructibility interesting because it makes little sense to adopt a memory model if any implementation of it must maintain a stronger model. One important result of this chapter is that such a stronger model is unique. We prove that for any memory model Δ, the class of constructible memory models stronger than Δ has a unique weakest element, which we call the ***constructible version*** Δ^* of Δ.

We discuss two approaches for specifying memory models within this theory. In the first approach, a memory model is defined in terms of topological sorts of the computation. Using this approach, we generalize the definition of ***sequential consistency*** [96], and redefine the ***location consistency*** model from Chapter 4,[1] in which every location is serialized independently of other locations. In the second approach, a memory model is defined by imposing certain constraints on the

[1]Location consistency is often called coherence in the literature [79]. It is *not* the model with the same name introduced by Gao and Sarkar [61]. See [54] for a justification of this terminology.

value that the observer function can assume on paths in the computation dag. Using this approach, we explore the class of ***dag-consistent*** memory models, a generalization of the ***dag consistency*** of [27, 26, 85]. Such models do not even require that a single location be serialized, and are thus strictly weaker than the other class of models. Nonetheless, we found an interesting link between location consistency, dag consistency and constructibility. The strongest variant of dag consistency (called ***NN-dag consistency***) is not constructible, and is strictly weaker than location consistency. Its constructible version, however, turns out to be the same model as location consistency.

We believe that the advantages of the computation-centric framework transcend the particular results mentioned so far. First, we believe that reasoning about computations is easier than reasoning about processors. Second, the framework is completely formal, and thus we can make rigorous proofs of the correctness of a memory. Third, our approach allows us to generalize familiar memory models, such as sequential consistency. Most of the simplicity of our theory comes from ignoring the fundamental issue of how programs generate computations. This simplification does not come without cost, however. The computation generated by a program may depend on the values received from the memory, which in turn depend on the computation. It remains important to account for this circularity within a unified theory. We believe, however, that the problem of memory semantics alone is sufficiently difficult that it is better to isolate it initially.

The rest of this chapter is organized as follows. In Section 5.1, we present the basic computation-centric theory axiomatically. In Section 5.2, we define constructibility, prove the uniqueness of the constructible version, and establish necessary and sufficient conditions for constructibility to hold. In Section 5.3, we discuss models based on a topological sort, and give computation-centric definitions of sequential consistency [96] and location consistency. In Section 5.4, we define the class of dag-consistent memory models and investigate the relations among them. In Section 5.5, we prove that location consistency is the constructible version of NN-dag consistency. Finally, we situate our work in the context of related research in Section 5.6.

5.1 Computation-centric memory models

In this section, we define the basic concepts of the computation-centric theory of memory models. The main definitions are those of a ***computation*** (Definition 31), an ***observer function*** (Definition 32), and a ***memory model*** (Definition 33). We also define two straightforward properties of memory models called ***completeness*** and ***monotonicity***.

We start with a formal definition of memory. A ***memory*** is characterized by a set $\mathcal{L}$ of ***locations***, a set $\mathcal{O}$ of abstract instructions (such as read and write), and a set of ***values*** that can be stored at each location. In the rest of the chapter, we abstract away the actual data, and consider a memory to be characterized by $\mathcal{L}$ and $\mathcal{O}$, using values only for concrete examples.

For a set $\mathcal{O}$ of abstract instructions, we formally define a computation as follows.

Definition 31 *A **computation** $C = (\mathcal{G}, op)$ is a pair of a finite directed acyclic graph (dag) $\mathcal{G} = (V, E)$ and a function $op : V \mapsto \mathcal{O}$.*

For a computation C, we use $\mathcal{G}_C$, V_C, E_C and op_C to indicate its various components. The smallest computation is the ***empty computation*** ε, which has an empty dag. Intuitively, each node $u \in V$ represents an instance of the instruction $op(u)$, and each edge indicates a dependency between its endpoints.

The way a computation is generated from an actual execution depends on the language used to write the program. For example, consider a program written in a language with fork/join parallelism. The execution of the program can be viewed as a set of operations on memory that obey the dependencies imposed by the fork/join constructs. The issues of how the computation is expressed and scheduled are extremely important, but in this chapter, we consider the computation as fixed and given *a priori*. The Cilk system demonstrates one way to address the scheduling problem.

In this chapter, we consider only read-write memories. We denote reads and writes to location l by $R(l)$ and $W(l)$ respectively. For the rest of the chapter, the set of instructions is assumed to be $\mathcal{O} = \{R(l) : l \in \mathcal{L}\} \cup \{W(l) : l \in \mathcal{L}\} \cup \{N\}$, where N denotes any instruction that does not access the memory (a "no-op").

We now define some terminology for dags and computations. If there is a path from node u to node v in the dag $\mathcal{G}$, we say that u ***precedes*** v in $\mathcal{G}$, and we write $u \preceq_{\mathcal{G}} v$. We may omit the dag and write $u \preceq v$ when it is clear from context. We often need to indicate strict precedence, in which case we write $u \prec v$. A ***relaxation*** of a dag $\mathcal{G} = (V, E)$ is any dag (V, E') such that $E' \subseteq E$. A ***prefix*** of $\mathcal{G}$ is any subgraph $\mathcal{G}' = (V', E')$ of $\mathcal{G}$ such that if $(u, v) \in E$ and $v \in V'$, then $u \in V'$ and $(u, v) \in E'$.

A ***topological sort*** T of $\mathcal{G} = (V, E)$ is a total order on V consistent with the precedence relation, i.e., $u \preceq_{\mathcal{G}} v$ implies that u precedes v in T. The precedence relation of the topological sort is denoted with $u \preceq_T v$. We represent topological sorts as sequences, and denote by $\mathcal{TS}(\mathcal{G})$ the set of all topological sorts of a dag $\mathcal{G}$. Note that for any $V' \subseteq V$, if $\mathcal{G}'$ is the subgraph of $\mathcal{G}$ induced by V' and $\mathcal{G}''$ is the subgraph induced by $V - V'$, and T' and T'' are topological sorts of $\mathcal{G}'$ and $\mathcal{G}''$ respectively, then the concatenation of T' and T'' is a topological sort of $\mathcal{G}$ if and only if for all $u \in V'$ and $v \in V - V'$, we have $v \not\preceq_{\mathcal{G}} u$.

For a computation $C = (\mathcal{G}, op)$, if $\mathcal{G}'$ is a subgraph of $\mathcal{G}$ and op' is the restriction of op to $\mathcal{G}'$, then $C' = (\mathcal{G}', op')$ is a ***subcomputation*** of C. We also call op' the ***restriction*** of op to C', and denote it by $op|_{C'}$, i.e., $op|_{C'}(u) = op(u)$ for all $u \in V_{C'}$. We abuse notation by using the same terminology for computations as for dags. For example, C' is a ***prefix*** of C if $\mathcal{G}_{C'}$ is a prefix of $\mathcal{G}_C$ and $op_{C'} = op_C|_{C'}$. Similarly, $\mathcal{TS}(C) = \mathcal{TS}(\mathcal{G}_C)$. In addition, C is an ***extension*** of C' by $o \in \mathcal{O}$ if C' is a prefix of C, $V_C = V_{C'} \cup \{u\}$ for some $u \notin V_{C'}$ and $op_C(u) = o$. Note that if C' is a prefix of C with $|V_C| = |V_{C'}| + 1$ then C is an extension of C' by $op_C(u)$, where $u \in V_C - V_{C'}$.

We imagine a computation as being executed in some way by one or more processors, subject to the dependency constraints specified by the dag, and we want to define precisely the semantics of the read and write operations. For this purpose, rather than specifying the meaning of read and write operations directly, we introduce a technical device called an ***observer function***. For every node u in the computation and for every location l, the value of the observer function $v = \Phi(l, u)$ is another node that writes to l. The idea is that u "observes" the write performed by v, so that if u reads l, it receives the value written by v. The observer function can assume the special value $\perp$, indicating that no write has been observed, in which case a read operation receives an undefined value. Note that $\perp$ is not a value stored at a location, but an element of the range of the observer function similar to a node of the computation. For notational convenience, we extend the precedence relation so that $\perp \prec u$ for every node u of any computation, and we also include $\perp$ as a node in the domain of observer functions.

Definition 32 *An **observer function** for a computation C is a function $\Phi : \mathcal{L} \times V_C \cup \{\perp\} \mapsto V_C \cup \{\perp\}$ satisfying the following properties for all $l \in \mathcal{L}$ and $u \in V_C \cup \{\perp\}$:*

32.1. If $\Phi(l, u) = v \neq \perp$ then $op_C(v) = W(l)$.

32.2. $u \not\prec \Phi(l, u)$.

32.3. If $u \neq \perp$ and $op_C(u) = W(l)$ then $\Phi(l, u) = u$.

Informally, every observed node must be a write (part 32.1), and a node cannot precede the node it observes (part 32.2). Furthermore, every write must observe itself (part 32.3). Note that Condition 32.2 implies $\Phi(l, \perp) = \perp$ for all $l \in \mathcal{L}$. The empty computation has a unique observer function, which we denote by Φ_ε.

The observer function allows us to abstract away from memory values, and to give memory semantics even to nodes that do not perform memory operations. In other words, our formalism may distinguish two observer functions that produce the same execution. We choose this formalism because it allows a computation node to denote some form of synchronization, which affects the memory semantics even if the node does not access the memory.

A ***memory model*** Δ is a set of pairs of computations and observer functions, including the empty computation and its observer function,[2] as stated formally by the next definition.

Definition 33 *A **memory model** is a set Δ such that*

$$\{(\varepsilon, \Phi_\varepsilon)\} \subseteq \Delta \subseteq \{(C, \Phi) : \Phi \textit{ is an observer function for } C\}$$

The next definition is used to compare memory models.

[2] This is a technical requirement to simplify boundary cases.

Definition 34 *A model Δ is **stronger** than a model Δ' if $\Delta \subseteq \Delta'$. We also say that Δ' is **weaker** than Δ.*

Notice that the subset, not the superset, is said to be stronger, because the subset allows fewer memory behaviors.

A memory model may provide an observer function only for some computations. It is natural to restrict ourselves to those models that define at least one observer function for each computation. We call such models complete. Formally, a memory model Δ is ***complete*** if, for every computation C, there exists an observer function Φ such that $(C, \Phi) \in \Delta$.

From the definitions of weaker and complete, it follows that any model weaker than some complete model is also complete. Formally, if Δ is complete and $\Delta' \supseteq \Delta$, then Δ' is also complete.

Another natural property for memory models to satisfy is that relaxations of a computation should not invalidate observer functions for the original computation. We call this property monotonicity.

Definition 35 *A memory model Δ is **monotonic** if for all $(C, \Phi) \in \Delta$, we also have $(C', \Phi) \in \Delta$, for all relaxations C' of C.*

Monotonicity is a technical property that simplifies certain proofs (for example, see Theorem 42), and we regard it as a natural requirement for any "reasonable" memory model.

5.2 Constructibility

In this section, we define a key property of memory models that we call ***constructibility***. Constructibility says that if we have a computation and an observer function in some model, it is always possible to extend the observer function to a "bigger" computation. Not all memory models are constructible. However, there is a natural way to define a unique ***constructible version*** of a nonconstructible memory model. At the end of the section, we give a necessary and sufficient condition for the constructibility of monotonic memory models.

The motivation behind constructibility is the following. Suppose that, instead of being given completely at the beginning of an execution, a computation is revealed one node at a time by an adversary.[3] Suppose also that there is an algorithm that maintains a given memory model online. Intuitively, the algorithm constructs an observer function as the computation is revealed. Suppose there is some observer function for the part of the computation revealed so far, but when the adversary reveals the next node, there is no way to assign a value to it that satisfies the memory model. In this case, the consistency algorithm is "stuck". It should have chosen a different observer function in the past, but that would have required some knowledge of the future. Constructibility says that

[3]This is the case with multithreaded languages such as Cilk.

this situation cannot happen: if Φ is a valid observer function in a constructible model, then there is always a way to extend Φ to a "bigger" computation as it is revealed.

Definition 36 *A memory model Δ is **constructible** if the following property holds: for all computations C' and for all prefixes C of C', if $(C, \Phi) \in \Delta$ then there exists an observer function Φ' for C' such that $(C', \Phi') \in \Delta$ and the restriction of Φ' to C is Φ, i.e., $\Phi'|_C = \Phi$.*

Completeness follows immediately from constructibility, since the empty computation is a prefix of all computations and, together with its unique observer function, belongs to every memory model.

Not all memory models are constructible; we shall discuss some nonconstructible memory models in Section 5.4. However, a nonconstructible model Δ can be strengthened in an essentially unique way until it becomes constructible. More precisely, the set of constructible models stronger than Δ contains a unique weakest element Δ^*, which we call the ***constructible version*** of Δ. To prove this statement, we first prove that the union of constructible models is constructible.

Lemma 37 *Let $\mathcal{S}$ be a (possibly infinite) set of constructible memory models. Then $\bigcup_{\Delta \in \mathcal{S}} \Delta$ is constructible.*

Proof: Let C' be a computation and C be a prefix of C'. We must prove that, if $(C, \Phi) \in \bigcup_{\Delta \in \mathcal{S}} \Delta$, then an extension Φ' of the observer function Φ exists such that $(C', \Phi') \in \bigcup_{\Delta \in \mathcal{S}} \Delta$.

If $(C, \Phi) \in \bigcup_{\Delta \in \mathcal{S}} \Delta$ then $(C, \Phi) \in \Delta$ for some $\Delta \in \mathcal{S}$. Since Δ is constructible, there exists an observer function Φ' for C' such that $(C', \Phi') \in \Delta$ and $\Phi'|_C = \Phi'$. Thus, $(C', \Phi') \in \bigcup_{\Delta \in \mathcal{S}} \Delta$, as required. ■

We now define the constructible version of a model Δ, and prove that it is the weakest constructible model stronger than Δ.

Definition 38 *The **constructible version** Δ^* of a memory model Δ is the union of all constructible models stronger than Δ.*

Theorem 39 *For any memory model Δ,*

39.1. $\Delta^* \subseteq \Delta$*;*

39.2. Δ^* *is constructible;*

39.3. for any constructible model Δ' such that $\Delta' \subseteq \Delta$, we have $\Delta' \subseteq \Delta^*$.

Proof: Δ^* satisfies Conditions 39.1 and 39.3 by construction, and Condition 39.2 because of Lemma 37. ■

In two theorems, we establish conditions that guarantee constructibility. Theorem 40 gives a sufficient condition for the constructibility of general memory models. For monotonic memory models, the condition is simpler (Theorem 42).

Theorem 40 *A memory model Δ is constructible if for any $(C, \Phi) \in \Delta$, $o \in \mathcal{O}$, and extension C' of C by o, there exists an observer function Φ' for C' such that $(C', \Phi') \in \Delta$ and $\Phi = \Phi'|_C$.*

Proof: We must prove that if C is a prefix of C' and $(C, \Phi) \in \Delta$, then there exists an observer function Φ' for C' such that $(C', \Phi') \in \Delta$ and $\Phi'|_C = \Phi$.

Since C is a prefix of C', there exists a sequence of computations $C_0, C_1, \ldots, C_k$ such that $C_0 = C$, $C_k = C'$, and C_i is an extension of C_{i-1} by some $o_i \in \mathcal{O}$ for all $i = 1, \ldots, k$, where $k = |V_{C'}| - |V_C|$.

The proof of the theorem is by induction on k. The base case $k = 0$ is trivial since $C' = C$. Now, suppose inductively that there exists Φ_{k-1} such that $(C_{k-1}, \Phi_{k-1}) \in \Delta$. Since C' is an extension of C_{k-1} by o_k, the theorem hypothesis implies that an observer function Φ' exists such that $(C', \Phi') \in \Delta$, as required to complete the inductive step. ■

For monotonic memory models, we do not need to check every extension of a computation to prove constructibility, but rather only a small class of them, which we call the ***augmented computations***. An augmented computation is an extension by one "new" node, where the "new" node is a successor of all "old" nodes.

Definition 41 *Let C be a computation and $o \in \mathcal{O}$ be any operation. The* ***augmented computation*** *of C by o, denoted $aug_o(C)$, is the computation C' such that*

$$
\begin{aligned}
V_{C'} &= V_C \cup \{\mathit{final}(C)\} \\
E_{C'} &= E_C \cup \{(v, \mathit{final}(C)) : v \in V_C\} \\
op_{C'}(v) &= \begin{cases} op_C(v) & \text{for } v \in V_C \\ o & \text{for } v = \mathit{final}(C) \end{cases},
\end{aligned}
$$

where $\mathit{final}(C) \notin V_C$ is a new node.

The final theorem of this section states that if a monotonic memory model can extend the observer function for any computation to its augmented computations, then the memory model is constructible.

Theorem 42 *A monotonic memory model Δ is constructible if and only if for all $(C, \Phi) \in \Delta$ and $o \in \mathcal{O}$, there exists an observer function Φ' such that $(aug_o(C), \Phi') \in \Delta$ and $\Phi'|_C = \Phi$.*

Proof: The "⇒" part is obvious, since C is a prefix of $aug_o(C)$.

For the "$\Leftarrow$" direction, suppose $(C, \Phi) \in \Delta$ and $o \in \mathcal{O}$. By hypothesis, there exists Φ' such that $(aug_o(C), \Phi') \in \Delta$. For any extension C' of C by o, note that C' is a relaxation of $aug_o(C)$. Since Δ is monotonic, we also have $(C', \Phi') \in \Delta$. Thus, by Theorem 40, Δ is constructible. ∎

One interpretation of Theorem 42 is the following. Consider an execution of a computation. At any point in time some prefix of the computation will have been executed. If at all times it is possible to define a "final" state of the memory (given by the observer function on the final node of the augmented computation) then the memory model is constructible.

5.3 Models based on topological sorts

In this section, we define two well known memory models in terms of topological sorts of a computation. The first model is ***sequential consistency*** [96]. The second model is sometimes called ***coherence*** in the literature [61, 79]; we call it ***location consistency***. Both models are complete, monotonic and constructible. Because we define these models using computations, our definitions generalize traditional processor-centric ones without requiring explicit synchronization operations.

It is convenient to state both definitions in terms of the "last writer preceding a given node", which is well defined if we superimpose a total order on a computation, producing a topological sort.

Definition 43 *Let C be a computation, and $T \in \mathcal{TS}(C)$ be a topological sort of C. The* ***last writer function*** *according to T is $\mathcal{W}_T : \mathcal{L} \times V_C \cup \{\bot\} \mapsto V_C \cup \{\bot\}$ such that for all $l \in \mathcal{L}$ and $u \in V_C \cup \{\bot\}$:*

43.1. If $\mathcal{W}_T(l, u) = v \neq \bot$ then $op_C(v) = W(l)$.

43.2. $\mathcal{W}_T(l, u) \preceq_T u$.

43.3. $\mathcal{W}_T(l, u) \prec_T v \preceq_T u \implies op_C(v) \neq W(l)$ for all $v \in V_C$.

We now prove two straightforward facts about last writer functions. The first states that Definition 43 is well defined. The second states that if w is the last writer preceding a node u, then it is also the last writer preceding any node between w and u.

Theorem 44 *For any topological sort T, there exists a unique last writer function according to T.*

Proof: It is sufficient to show that for any $l \in \mathcal{L}$ and $u \in V_C$, there is a unique $v \in V_C \cup \{\bot\}$ such that $\mathcal{W}_T(l, u) = v$ satisfies the three conditions in the definition of $\mathcal{W}_T$.

Suppose that v and v' both satisfy these conditions. Since T is a topological sort, we assume without loss of generality that $v \preceq_T v'$. If $v' = \bot$ then $v = \bot$. Otherwise, using $v' = \mathcal{W}_T(l, u)$ in Conditions 43.1 and 43.2, $op_C(v') = W(l)$ and $v' \preceq_T u$. Thus, using $v = \mathcal{W}_T(l, u)$ in Condition 43.3, we get $v \not\prec_T v'$. In either case, $v = v'$ as required. ∎

Theorem 45 *For any computation C, if $\mathcal{W}_T$ is the last writer function according to T for some $T \in \mathcal{TS}(C)$ then for all $u, v \in V_C$ and $l \in \mathcal{L}$ such that $\mathcal{W}_T(l,u) \prec_T v \preceq_T u$, we have $\mathcal{W}_T(l,v) = \mathcal{W}_T(l,u)$.*

Proof: Let $w = \mathcal{W}_T(l,u)$. Because of Theorem 44, it is sufficient to prove that w satisfies the three conditions for $\mathcal{W}_T(l,v)$. It satisfies Condition 43.2 by hypothesis, and it satisfies Condition 43.1 since it is the last writer preceding u. Finally, note that any v' such that $w \prec_T v' \preceq_T v$ also satisfies $w \prec_T v' \preceq_T u$, so by Condition 43.3 applied to u, $op_C(v') \neq W(l)$. Thus, $\mathcal{W}_T(l,v) = w = \mathcal{W}_T(l,u)$. ∎

We use the last writer function for defining memory models, which is possible because the the last writer function is an observer function, as stated in the next theorem.

Theorem 46 *Let C be a computation, and $T \in \mathcal{TS}(C)$ be a topological sort of C. The last writer function $\mathcal{W}_T$ is an observer function for C.*

Proof: Condition 43.1 is the same as Condition 32.1 and Condition 32.2 is implied by Condition 43.2. Finally, note that the contrapositive of Condition 43.3 with $v = u \neq \perp$ is $op_C(u) = W(l) \implies \mathcal{W}_T(l,u) \not\prec_T u$. Using Condition 43.2, this simplifies to $op_C(u) = W(l) \implies \mathcal{W}_T(l,u) = u$, thus proving Condition 32.3. ∎

We define sequential consistency using last writer functions.

Definition 47 ***Sequential consistency*** *is the memory model*

$$SC = \{(C, \mathcal{W}_T) : T \in \mathcal{TS}(C)\}$$

This definition captures the spirit of Lamport's original model [96], that there exists a global total order of events observed by all nodes. However, unlike Lamport's definition, it does not restrict dependencies to be sequences of operations at each processor, nor does it depend on how the computation is mapped onto processors.

Sequential consistency requires that the topological sort be the same for all locations. By allowing a different topological sort for each location, we define a memory model that is often called ***coherence*** [61, 79]. We believe that a more appropriate name for this model is ***location consistency***, even though the same name is used in [61] for a different memory model.[4]

Definition 48 ***Location consistency*** *is the memory model*

$$LC = \{(C, \Phi) : \forall l \; \exists T_l \in \mathcal{TS}(C) \; \forall u, \; \Phi(l,u) = \mathcal{W}_{T_l}(l,u)\}$$

[4] See [54] for a discussion of this terminology.

Location consistency requires that all writes to the same location behave *as if* they were serialized. This need not be the case in the actual implementation. For example, the BACKER algorithm from [27, 26] maintains location consistency, even though it may keep several incoherent copies of the same location. In Section 5.5, we prove that location consistency is the constructible version of a model we call NN-dag consistency.

It follows immediately from the definitions that SC is stronger than LC. In fact, this relation is strict as long as there is more than one location.

Both SC and LC are complete memory models, because an observer function can be constructed for any computation by sorting the dag and using the last writer function. We now prove that they are also monotonic and constructible.

Theorem 49 *SC and LC are monotonic and constructible memory models.*

Proof: The monotonicity of both follows immediately from the definition since $\mathcal{TS}(C) \subseteq \mathcal{TS}(C')$ for all relaxations C' of C.

For constructibility, we give only the proof for SC; the proof for LC is similar. Since SC is monotonic, we only need to prove that it is possible to extend any observer function for a computation to its augmented computation, and then apply Theorem 42.

If $(C, \Phi) \in$ SC then, by definition of SC, $\Phi = \mathcal{W}_T$ for some topological sort $T \in \mathcal{TS}(C)$. For each $o \in \mathcal{O}$, consider the augmented computation $\mathit{aug}_o(C)$, and let T' be the following total order of the nodes of $\mathit{aug}_o(C)$: all the nodes of C in T order, followed by $\mathit{final}(C)$. It is immediate that T' is a topological sort of $\mathit{aug}_o(C)$. Thus, $\mathcal{W}_{T'}$ is a valid SC observer function for $\mathit{aug}_o(C)$, and $\mathcal{W}_{T'}|_C = \mathcal{W}_T = \Phi$. The conclusion follows by application of Theorem 42. ∎

5.4 Dag-consistent memory models

In this section, we consider the class of ***dag-consistent*** memory models, which are not based on topological sorts of the computation. Rather, dag-consistent models impose conditions on the value that the observer function can assume on paths in the computation. We focus on four "interesting" dag-consistent memory models, and investigate their mutual relations.

In the dag-consistent models the observer function obeys a restriction of the following form: If a node lies on a path between two other nodes, and the observer function assumes the value x at the two end nodes, and the three nodes satisfy certain additional conditions, then the observer function also assumes the value x at the middle node. The various dag consistency models differ in the additional conditions they impose on the nodes.

Definition 50 *Let Q be a predicate on $\mathcal{L} \times V \times V \times V$, where V is the set of all nodes of a computation. The Q-**dag consistency** memory model is the set of all pairs (C, Φ) such that Φ is an observer function for C and the following condition holds:*

50.1. For all locations $l \in \mathcal{L}$ and nodes $u, v, w \in V_C \cup \{\bot\}$ such that $u \prec v \prec w$ and $Q(l, u, v, w)$, we have $\Phi(l, u) = \Phi(l, w) \implies \Phi(l, v) = \Phi(l, u)$.

Definition 50 is a generalization of the two definitions of dag consistency that the Cilk group of MIT (including myself) proposed in the past [27, 26]. Varying the predicate Q in Condition 50.1 yields different memory models. Note that strengthening Q weakens the memory model.

In the rest of the chapter, we consider four specific predicates, NN, NW, WN and WW, and the dag consistency models they define. These predicates do not depend on w, but only on whether u and v write to l. The rationale behind the names is that "W" stands for "write", and "N" stands for "do not care". For example, WN means that the first node is a write and we do not care about the second. Formally,

$$\begin{aligned}
\text{NN}(l, u, v, w) &= \textit{true} \\
\text{NW}(l, u, v, w) &= \text{``}op_C(v) = W(l)\text{''} \\
\text{WN}(l, u, v, w) &= \text{``}op_C(u) = W(l)\text{''} \\
\text{WW}(l, u, v, w) &= \text{NW}(l, u, v, w) \wedge \text{WN}(l, u, v, w)
\end{aligned}$$

We use NN as a shorthand for NN-dag consistency, and similarly for WN, NW and WW.

The relations among NN, WN, NW, WW, LC and SC are shown in Figure 5-1. WW is the original dag consistency model defined in [27, 85]. WN is the model called dag consistency in [26], strengthened to avoid anomalies such as the one illustrated in Figure 5-2. NN is the strongest dag-consistent memory model (as proven in Theorem 51 below). Symmetry suggests that we also consider NW.

Theorem 51 *NN $\subseteq$ Q-dag consistency for any predicate Q.*

Proof: The proof is immediate from the definition: an observer function satisfying Condition 50.1 with $Q(l, u, v, w) = \textit{true}$ will satisfy Condition 50.1 for any other predicate Q. ∎

The rest of the chapter is mostly concerned with the proof of the relations shown in Figure 5-1. We have already observed in Section 5.3 that SC is strictly stronger than LC. In the rest of this section, we give informal proofs of the relations among the dag-consistent models. Proving relations between the dag-consistent models and the models based on topological sorts, however, is more involved, and we postpone the proof that LC $\subsetneq$ NN and that LC = NN* until Section 5.5.

That NN $\subseteq$ NW $\subseteq$ WW and NN $\subseteq$ WN $\subseteq$ WW follows immediately from the definitions of these models. To see that these inclusions are strict and that WN $\not\subseteq$ NW and NW $\not\subseteq$ WN, consider

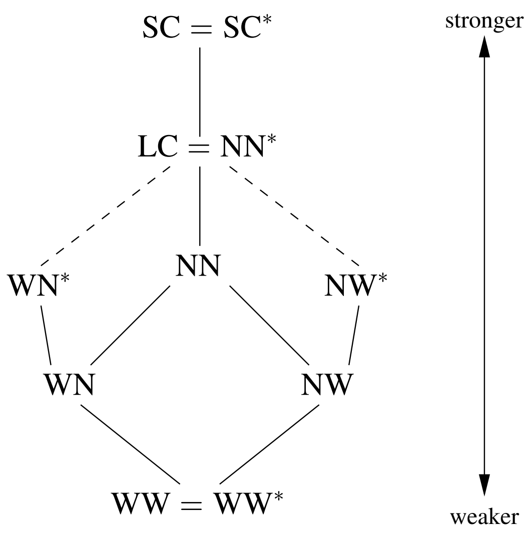


Figure 5-1: The relations among (some) dag-consistent models. A straight line indicates that the model at the lower end of the line is strictly weaker than the model at the upper end. For example, LC is strictly weaker than SC. It is known that LC $\subseteq$ WN* and that LC $\subseteq$ NW*, but we do not know whether these inclusions are strict. This situation is indicated with a dashed line.

the computation/observer function pairs shown in Figures 5-2 and 5-3. These examples illustrate operations on a single memory location, which is implicit. It is easy to verify that the first pair is in WW and NW but not WN and NN, and the second is in WW and WN but not NW and NN. We could also show that NN $\subsetneq$ NW $\cap$ WN and WW $\supsetneq$ NW $\cup$ WN, using similar examples.

To see that NN is not constructible, let C' be the computation in Figure 5-4, and (C, Φ) be the computation/observer function pair to the left of the dashed line. It is easy to verify that C is a prefix of C' and that $(C, \Phi) \in$ NN. However, unless F writes to the memory location, there is no way to extend Φ to C' without violating NN-dag consistency. Formally, there is no Φ' such that $(C', \Phi') \in$ NN and $\Phi'|_C = \Phi$. Informally, suppose that we use an algorithm that claims to support NN-dag consistency. The adversary reveals the computation C, and our algorithm produces the observer function Φ, which satisfies NN-dag consistency. Then the adversary reveals the new node F. The algorithm is "stuck"; it cannot assign a value to the observer function for F that satisfies NN-dag consistency.

The same example shows that WN is not constructible, and a similar one can be used to show that NW is not constructible. WW is constructible, although we do not prove this fact in this dissertation.

Historically, we investigated the various dag-consistent models after discovering the problem with WN illustrated in Figure 5-4. Our attempts to find a "better" definition of dag consistency led us to the notion of constructibility. As Figure 5-1 shows, among the four models only WW is constructible. A full discussion of these models (including a criticism of WW) can be found in [54].

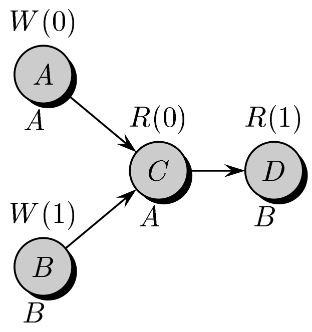


Figure 5-2: An example of a computation/observer function pair in WW and NW but not WN or NN. The computation has four nodes, A, B, C and D (the name of the node is shown inside the node). The memory consists of a single location, which is implicit. Every node performs a read or a write operation on the location, and this is indicated above the node. For example, $W(0)$ means that the node writes a 0 to the location, and $R(1)$ means that it reads a 1. The value of the observer function is displayed below each node. For example, the value of the function for node C is A, which accounts for the fact that node C reads the value written by node A.

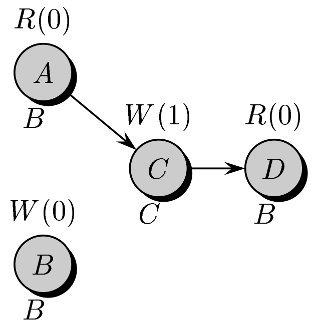


Figure 5-3: An example of a computation/observer function pair in WW and WN but not NW or NN. The conventions used in this figure are explained in Figure 5-2.

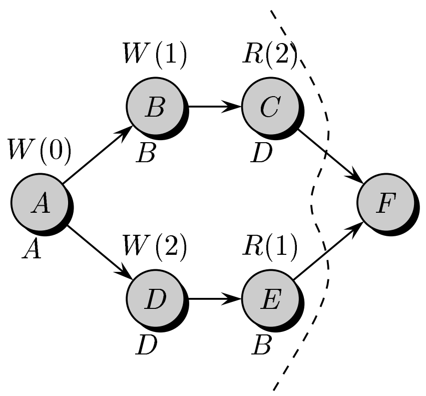


Figure 5-4: An example demonstrating the nonconstructibility of NN. The conventions used in this figure are explained in Figure 5-2. A new node F has been revealed by the adversary after the left part of the computation has been executed. It is not possible to assign a value to the observer function for node F satisfying NN-dag consistency.

At this stage of our research, little is known about WN* and NW*, which would be alternative ways of defining dag consistency.

5.5 Dag consistency and location consistency

In this section, we investigate the relation between NN-dag consistency and location consistency. We show that location consistency is strictly stronger than any dag-consistent model, and moreover, that it is the constructible version of NN-dag consistency, i.e., LC $=$ NN*.

We begin by proving that LC is strictly stronger than NN, which implies that NN* is no stronger than LC, since LC is constructible.

Theorem 52 *LC $\subsetneq$ NN.*

Proof: We first prove that LC $\subseteq$ NN. Let $(C, \Phi) \in$ LC. We want to prove that $(C, \Phi) \in$ NN. For each location l, we argue as follows: By the definition of LC, there exists $T \in \mathcal{TS}(C)$ such that $\mathcal{W}_T(l, u) = \Phi(l, u)$ for all $u \in V$.

Suppose that $u \prec v \prec w$ and $\Phi(l, u) = \Phi(l, w)$. Then $\mathcal{W}_T(l, w) = \mathcal{W}_T(l, u) \preceq_T u \prec_T v \prec_T w$. So by Theorem 45, $\mathcal{W}_T(l, v) = \mathcal{W}_T(l, u)$. Thus $\Phi(l, v) = \Phi(l, u)$ as required.

To complete the proof, we only need to note that LC $\neq$ NN since LC is constructible and NN is not. ∎

From Theorems 51 and 52, it immediately follows that LC is strictly stronger than any dag-consistent memory model. And since LC is complete, it follows from that all dag-consistent models are complete.

Finally, we prove that the constructible version of NN-dag consistency is exactly location consistency.

Theorem 53 *LC $=$ NN*.*

Proof: We first prove that NN* $\supseteq$ LC, and then that NN* $\subseteq$ LC. By Theorem 52, LC $\subseteq$ NN, and by Theorem 49, LC is constructible. Therefore, by Condition 39.3, we have that NN* $\supseteq$ LC. That NN* $\subseteq$ LC is implied by the claim that follows.

Claim: For any nonnegative integer k, suppose $(C, \Phi) \in$ NN* and $|V_C| = k$. Then for each $l \in \mathcal{L}$, there exists $T \in \mathcal{TS}(C)$ such that $\Phi(l, u) = \mathcal{W}_T(l, u)$, for all $u \in V_C$.

Proof of claim: The proof is by strong induction on k. The claim is trivially true if $k = 0$, since $C = \varepsilon$ and $\Phi = \Phi_\varepsilon$ in this case.

If $k > 0$, assume inductively that the claim is true for all computations with fewer than k nodes. We prove it is true for C. Since NN* is constructible, Theorem 42 implies that there exists Φ' such that $(aug_N(C), \Phi') \in$ NN* and $\Phi'|_C = \Phi$. There are two cases: either $\Phi'(l, \mathit{final}(C)) = \perp$ or not.

If $\Phi'(l, \mathit{final}(C)) = \perp$ then, by the definition of NN, $\Phi(l, u) = \perp$ for all $u \in V_C$ since $\perp \prec u \prec \mathit{final}(C)$. Thus, by Condition 32.3, $op_C(u) \neq W(l)$ for all $u \in V_C$. Thus, for any $T \in \mathcal{TS}(C)$, $\mathcal{W}_T(l, u) = \perp$ for all $u \in V_C$, as required.

Otherwise, let $w = \Phi'(l, \mathit{final}(C)) \in V_C$, let C' be the subcomputation of C induced by $\{u \in V_C : \Phi(l, u) \neq w\}$, and let C'' be the subcomputation of C induced by $\{u \in V_C : \Phi(l, u) = w\}$. That is, C' consists of nodes that do not observe w and C'' consists of nodes that observe w.

Since $w \notin V_{C'}$, we have $|V_{C'}| < k$, so by the inductive hypothesis, a topological sort $T' \in \mathcal{TS}(C')$ exists such that $\Phi(l, u) = \mathcal{W}_{T'}(l, u)$ for all $u \in V_{C'}$. Let T'' be any topological sort of C'' that begins with w; such a topological sort exists because $v \not\prec w$ for all $v \in V_{C''}$ by Condition 32.2. Since w is the only node of C'' that writes to l, $\mathcal{W}_{T''}(l, v) = w$ holds for all $v \in V_{C''}$. Let T be the concatenation of T' and T''. If we can prove that T is a legitimate topological sort of C, then the claim is proven, since $\mathcal{W}_T = \Phi$ by construction of T.

To prove that $T \in \mathcal{TS}(C)$, we only need to show that $v \not\prec u$ for all $u \in V_{C'}$ and $v \in V_{C''}$. This property holds, because otherwise $v \prec u \prec \mathit{final}(C)$, and by the NN-dag consistency property, $\Phi'(l, u) = \Phi'(l, v) = w$ must hold since $\Phi'(l, \mathit{final}(C)) = \Phi'(l, v) = w$. But this conclusion contradicts the assumption that $u \in V_{C'}$. ■

5.6 Discussion

This chapter presents a computation-centric formal framework for defining and understanding memory models. The idea that the partial order induced by a program should be the basis for defining memory semantics, as opposed to the sequential order of instructions within one processor, already appears in the work by Gao and Sarkar on their version of location consistency [61]. Motivated by the experience with dag consistency [27, 26, 85], we completely abstract away from a program, and assume the partial order (the "computation") as our starting point. *Post mortem* analysis has been used by [65] to verify (after the fact) that a given execution is sequentially consistent.

The need for formal frameworks for memory models has been felt by other researchers. Gibbons, Merrit, and Gharachorloo [67] use the I/O automata model of Lynch and Tuttle [105] to give a formal specification of release consistency [64]. Later work [66] extends the framework to non-blocking memories. The main concern of these papers is to expose the architectural assumptions that are implicit in previous literature on relaxed memory models. In this chapter, rather than focusing on the correctness of specific implementations of a memory model, we are more interested in the formal properties of models, such as constructibility.

A different formal approach has been taken by the proponents of the λ_S calculus [16], which is an extension of the λ calculus with synchronization and side-effects. The λ_S calculus gives a unified semantics of language *and* memory which is based on a set of rewriting rules. Preliminary λ_S

descriptions of sequential consistency [96] and location consistency (in the sense of Definition 48) exist [15].

Finally, many papers on memory models, starting with the seminal paper on sequential consistency [96], have been written from an hardware viewpoint, without a strict formal framework. The reader is referred to [79] and [2] for good tutorials and further references on the subject. Gharachorloo [63] also distinguishes *system-centric models*, which expose the programmer to the details of how a system may reorder operations, and *programmer-centric models*, which require the programmer to provide program-level information about the intended behavior of shared-memory operations but then allow the programmer to reason as if the memory were sequentially consistent. Both types of models, however, are processor-centric by our definition, since programs are still assumed to be sequential pieces of code running concurrently on several processors.

Historically, the abstract theory described in this chapter arose from concrete problems in the context of research on dag consistency, a memory model for the Cilk multithreaded language for parallel computing [28, 25, 85]. Dag consistency was developed to capture formally the minimal guarantees that users of Cilk expected from the memory. It was formulated to forbid particular behaviors considered undesirable when programming in Cilk. This point of view can be thought of as looking for the weakest "reasonable" memory model. (See [54] for a full discussion of this theme.) Dag consistency was also attractive because it is maintained by the BACKER algorithm used by Cilk, which has provably good performance [26].

Variants of dag consistency were developed to forbid "anomalies", or undesirable memory behaviors, as they were discovered. The papers [27] and [26] give two different definitions of dag consistency, which we call WW and WN. We were surprised to discover that WN is not constructible, and we tried both to find a "better" definition of dag consistency, and to capture the exact semantics of BACKER. Both problems have been solved. This chapter presents a more or less complete picture of the various dag-consistent models and their mutual relationships. In another paper, Luchangco [104] proves that BACKER supports location consistency. Consequently, the algorithmic analysis of [26] and the experimental results from [27] apply to location consistency with no change.

There are many possible directions in which this research can be extended. One obvious open problem is finding a simple characterization of NW* and WN*. It would also be useful to investigate whether any algorithm can be found that is more efficient than BACKER that implements a weaker memory model than LC. Another direction is to formulate other consistency models in the computation-centric framework. Some models, such as release consistency [64], require computations to be augmented with locks, and how to do this is a matter of active research. Finally, as mentioned previously, it is important to develop an integrated theory of memory and language semantics.

Chapter 6

FFTW

In previous chapters, we studied theoretical techniques for designing algorithms oblivious to the degree of parallelism and to the parameters of the cache. Real-world computer systems, however, are never completely described by any theory. For example, our previous discussion did not take into account details such as the structure of the processor pipeline, branch predictors, the limited associativity of caches, compiler transformations, and so on. We do not possess any accurate theory that predicts the behavior of the details of real-world processors and compilers. Because of this lack of theoretical understanding, we cannot design high-performance algorithms that are oblivious to the processor architecture in the same way as cache-oblivious algorithms are insensitive to the parameters of the cache. Nevertheless, in this chapter we study how to obtain portable high performance despite the intricacies of real systems.

To attain portable high performance in the face of diverse processor architectures, we adopt a "closed-loop," end-to-end approach. We do not attempt to model performance, but instead we allow a program to adapt itself to the processor architecture automatically. An example of such a self-optimizing program is the ***FFTW*** library that I have developed with Steven G. Johnson. FFTW (the *Fastest Fourier Transform in the West*) is a library of fast C routines for computing the discrete Fourier transform (DFT) in one or more dimensions, of both real and complex data, and of arbitrary input size. This chapter describes the mechanisms that FFTW uses to optimize itself and the `genfft` special-purpose compiler that generated 95% of the FFTW code.

The discrete Fourier transform (DFT) is arguably one of the most important computational problems, and it pervades most branches of science and engineering [121, 48]. For many practical applications it is important to have an implementation of the DFT that is as fast as possible. In the past, speed was the direct consequence of clever algorithms [48] that minimized the number of arithmetic operations. On present-day general-purpose microprocessors, however, the performance of a program is mostly determined by complicated interactions of the code with the processor architecture, and by the structure of the memory. Designing for performance under these conditions requires an intimate knowledge of the computer architecture and considerable effort. For example, [95] documents a case where adding a "no-op" instruction to a program doubles its speed because of a particular implementation of branch prediction.

The FFTW system copes with varying processor architecture by means of a self-optimizing approach, where the program itself adapts the computation to the details of the hardware. We have compared many C and Fortran implementations of the DFT on several machines, and our experiments show that FFTW typically yields significantly better performance than all other publicly available DFT software. More interestingly, while retaining complete portability, FFTW is competitive with or faster than proprietary codes such as Sun's Performance Library and IBM's ESSL library that are highly tuned for a single machine.

The mechanics of self-optimization is the following. In FFTW, the computation of the transform is accomplished by an ***executor*** that consists of highly optimized, composable blocks of C code called ***codelets***. A codelet is a specialized piece of code that computes part of the transform. For example, a codelet might compute a Fourier transform of a fixed size. The combination of codelets called by the executor is specified by a data structure called a ***plan***. The plan is determined at runtime, before the computation begins, by a ***planner*** which uses a dynamic programming algorithm [42, chapter 16] to find a fast composition of codelets. The planner tries to minimize the actual execution time, and not the number of floating point operations, since, as we shall see in Section 6.3, there is little correlation between these two performance measures. Consequently, the planner measures the run time of many plans and selects the fastest. In the current FFTW implementation, plans can also be saved to disk and used at a later time.

The speed of the executor depends crucially on the efficiency of the codelets, but writing and optimizing them is a tedious and error-prone process. We solve this problem in FFTW by means of ***metaprogramming***. Rather than being written by hand, FFTW's codelets are generated automatically by a special-purpose compiler called `genfft`. Written in the Objective Caml dialect of the functional language ML [99], `genfft` is a sophisticated program that first produces a representation of the codelet in the form of a data-flow graph, and then "optimizes" the codelet. In this optimization phase, `genfft` applies well-known transformations such as constant folding, and some DFT specific tricks (see Section 6.4.) Metaprogramming is a powerful technique for high-performance portability. First, a large space of codelets is essential for self-optimizing machinery to be effective. `genfft` produces many thousands of lines of optimized code—comparable in speed to what the best programmers could write by hand—within minutes. Second, it is easy to experiment with several algorithms and optimization strategies by changing only a handful lines of `genfft`'s code and regenerating the whole FFTW system. This experimentation process quickly converges to a high-performance implementation.

FFTW's internal sophistication is not visible to the user, however. The user interacts with FFTW only through the planner and the executor. (See Figure 6-1.) `genfft` is not used after compile time,

```
fftw_plan plan;
COMPLEX A[n], B[n];

/* plan the computation */
plan = fftw_create_plan(n);

/* execute the plan */
fftw(plan, A);

/* the plan can be reused for
   other inputs of size N */
fftw(plan, B);
```

Figure 6-1: Simplified example of FFTW's use for complex one-dimensional transform. The user must first create a plan, which can be then used at will. The same usage pattern applies to multidimensional transforms and to transforms of real data.

nor does the user need to know Objective Caml or have a Objective Caml compiler.[1] FFTW provides a function that creates a plan for a transform of a specified size, and once the plan has been created it can be used as many times as needed.

The FFTW library (currently at version 2.1.2) is free software available at the FFTW Web page.[2] FFTW is not a toy system, but a production-quality library that currently enjoys several thousand users and a few commercial customers. FFTW performs one- and multidimensional transforms, both of real and complex data, and it is not restricted to input sizes that are powers of 2. The distribution also contains parallel versions for Cilk-5 (see Chapter 2), Posix threads, and MPI [134].

While conceptually simple, the current FFTW system is complicated by the need of computing one- and multidimensional Fourier transforms of both complex and real data. The same pattern of planning and execution applies to all four modes of operation of FFTW: complex one-dimensional, complex multidimensional, real one-dimensional, and real multidimensional transforms. For simplicity, most of our discussion in this chapter focuses on one-dimensional Fourier transforms of complex data. In Section 6.8, we will see how FFTW uses similar ideas for the other kinds of transforms.

The rest of this chapter is organized as follows. Section 6.1 presents some background material on Fourier transforms. Section 6.2 presents experimental data that demonstrate FFTW's speed. Section 6.3 outlines the runtime structure of FFTW, consisting of the executor and the planner. The remaining sections are dedicated to `genfft`. Section 6.4 presents `genfft` at a high-level.

[1] In this sense, `genfft` resembles "Wittgenstein's ladder":

> My propositions are elucidatory in this way: he who understands me finally recognizes them as senseless, when he has climbed out through them, on them, over them. (He must so to speak throw away the ladder, after he has climbed up on it.) He must surmount these propositions; then he sees the world rightly.

(Approximate translation of [154, Proposition 6.54].)

[2] `http://theory.lcs.mit.edu/~fftw`

Section 6.5 describes what a codelet looks like when `genfft` constructs it. Section 6.6 describes how `genfft` optimizes a codelet. Section 6.7 describes the cache-oblivious scheduler that `genfft` uses to minimize the number of transfers between memory and registers. Section 6.8 discusses the implementation of real and multidimensional transforms. Section 6.9 discusses some pragmatic aspects of FFTW, such as `genfft`'s running time and memory requirements, the interaction of `genfft`'s output with C compilers, and the testing methodology that FFTW uses. Section 6.10 overviews related work on automatic generation of DFT programs.

6.1 Background

In this section we review some background material about the discrete Fourier transform (DFT). We give the definition of the DFT, and reference the most commonly used algorithms for computing it. See [48] for a more complete discussion.

Let X be an array of n complex numbers. The (one-dimensional, complex, forward) ***discrete Fourier transform*** of X is the array Y given by

$$Y[i] = \sum_{j=0}^{n-1} X[j]\omega_n^{-ij} \ , \tag{6.1}$$

where $\omega_n = e^{2\pi\sqrt{-1}/n}$ is a primitive n-th root of unity, and $0 \le i < n$. In case X is a real vector, the transform Y has the ***hermitian symmetry***

$$Y[n - i] = Y^*[i] \ ,$$

where $Y^*[i]$ is the complex conjugate of $Y^*[i]$.

The ***backward*** DFT flips the sign at the exponent of ω_n, and it is defined in the following equation.

$$Y[i] = \sum_{j=0}^{n-1} X[j]\omega_n^{ij} \ . \tag{6.2}$$

The backward transform is the "scaled inverse" of the forward DFT, in the sense that computing the backward transform of the forward transform yields the original array multiplied by n.

If n can be factored into $n = n_1 n_2$, Equation (6.1) can be rewritten as follows. Let $j = j_1 n_2 + j_2$, and $i = i_1 + i_2 n_1$. We then have,

$$Y[i_1 + i_2 n_1] = \sum_{j_2=0}^{n_2-1} \left[\left(\sum_{j_1=0}^{n_1-1} X[j_1 n_2 + j_2] \omega_{n_1}^{-i_1 j_1} \right) \omega_n^{-i_1 j_2} \right] \omega_{n_2}^{-i_2 j_2} . \tag{6.3}$$

This formula yields the ***Cooley-Tukey fast Fourier transform*** algorithm (FFT) [41]. The algorithm computes n_2 transforms of size n_1 (the inner sum), it multiplies the result by the so-called ***twiddle factors*** $\omega_n^{-i_1 j_2}$, and finally it computes n_1 transforms of size n_2 (the outer sum).

If $\gcd(n_1, n_2) = 1$, the ***prime factor*** algorithm can be applied, which avoids the multiplications by the twiddle factors at the expense of a more involved computation of indices. (See [121, page 619].) If n is a multiple of 4, the ***split-radix*** algorithm [48] can save some operations with respect to Cooley-Tukey. If n is prime, it is possible to use ***Rader's algorithm*** [126], which converts the transform into a circular convolution of size $n - 1$. The circular convolution can be computed recursively using two Fourier transforms, or by means of a clever technique due to Winograd [153] (FFTW does not employ Winograd's technique yet, however). Other algorithms are known for prime sizes, and this is still the subject of active research. See [144] for a recent compendium on the topic. Any algorithm for the forward DFT can be readily adapted to compute the backward DFT, the difference being that certain complex constants become conjugate. For the purposes of this chapter, we do not distinguish between forward and backward transform, and we simply refer to both as the "complex DFT".

In the case when the input is purely real, the transform can be computed with roughly half the number of operations of the complex case, and the hermitian output requires half the storage of a complex array of the same size. In general, keeping track of the hermitian symmetry throughout the recursion is nontrivial, however. This bookkeeping is relatively easy for the split-radix algorithm, and it becomes particularly nasty for the prime factor and the Rader algorithms. The topic is discussed in detail in [136]. In the real transform case, it becomes important to distinguish the forward transform, which takes a real input and produces an hermitian output, from the backward transform, whose input is hermitian and whose output is real, requiring a different algorithm. We refer to these cases as the "real to complex" and "complex to real" DFT, respectively.

The definition of DFT can be generalized to multidimensional input arrays. Informally, a multidimensional transform corresponds to transforming the input along each dimension. The precise order in which dimensions are transformed does not matter for complex transforms, but it becomes important for the real case, where one has to worry about which "half" array to compute in order to exploit the hermitian symmetry. We discuss these details in Section 6.8.

In the DFT literature, unlike in most of Computer Science, it is customary to report the exact number of arithmetic operations performed by the various algorithms, instead of their asymptotic complexity. Indeed, the time complexity of all DFT algorithms of interest is $O(n \log n)$, and a detailed count of the exact number of operation is usually doable (which by no means implies that the analysis is easy to carry out). It is no problem for me to follow this convention in this dissertation, because `genfft` produces the exact arithmetic complexity of a codelet.

In the literature, the term FFT ("fast Fourier transform") denotes either the Cooley-Tukey algorithm or any $O(n \log n)$ algorithm for the DFT, depending on the author. In this dissertation, FFT denotes any $O(n \log n)$ algorithm.

6.2 Performance results

This section present the result of benchmarking FFTW against many freely-available and a few proprietary codes. From the results of the benchmark, FFTW appears to be the fastest portable FFT implementation for most transform sizes. Indeed, its performance is competitive with that of the vendor-optimized Sun Performance and ESSL libraries on the UltraSPARC and the RS/6000, respectively.

Steven G. Johnson and I have benchmarked FFTW against about 50 other FFT programs written in the past 30 years (starting with Singleton's program [132] written in 1969), and we have collected performance results for one-, two-, and three-dimensional transforms on 10 different machines. Because of lack of space, we cannot include all these performance numbers here, but this selection of data should be sufficient to convince you that FFTW is both fast and portable. We show performance results from three machines: an IBM RS/6000 Model 3BT (120-MHz POWER2), a Sun HPC 5000 (167MHz UltraSPARC-I), and a DEC AlphaServer 4100 (467-MHz Alpha EV56). For each machine, we show performance results of both complex and real one-dimensional transforms in double precision. We show results for both the case where the input size is a power of 2, and for certain commonly used nonpowers of 2. (See Figures 6-2 through 6-13). For space reasons, for each machine we only show the performance of the 10 programs that execute fastest on average. Only 5 programs were available that compute real DFT's of size nonpower of 2, and the figures show all of them. The full collection of data, including multidimensional transforms, can be found at the FFTW web site.[3]

The performance results are given as a graph of the speed of the transform in MFLOPS versus array size. "MFLOPS" is a more-or-less arbitrary measure of performance, which can be thought of as the normalized inverse of execution time. For complex transforms, the MFLOPS count is computed by postulating the number of floating-point operations to be[4] $5n \lg n$, where n is the size of the input array. This is the operation count of the radix-2 Cooley-Tukey FFT algorithm

[3] `http://theory.lcs.mit.edu/~fftw`

[4] Recall that we use the notation $\lg x \triangleq \log_2 x$.

Bergland	A radix-8 C FFT, translated by Dr. Richard L. Lachance from a Fortran program by G. D. Bergland and M. T. Dolan. Works only for powers of 2, and does not include a true inverse transform. The original source can be found in [39].
Bernstein	A 1D C FFT (`djbfft` 0.60) by D. J. Bernstein (1997), optimized specifically for the Pentium and `gcc`. It is limited to transforms whose sizes are powers of 2 from 2 to 1024. This code is not strictly comparable to the rest of the programs since it produces out-of-order results.
Bloodworth	C FFT by Carey E. Bloodworth (1998), including real-complex transforms and fast Hartley transforms. Works only for powers of 2.
Crandall	C real-complex FFT by R. E. Crandall, developed as a part of a Mersenne-prime search program. Only works for powers of 2 and its output is in permuted order. See also [43].
CWP	A prime-factor FFT implementation by D. Hale in a C numerical library from the Colorado School of Mines.
***DXML**	FFT from the Digital Extended Math Library, optimized for the Alpha.
***ESSL**	IBM's ESSL library for the RS/6000.
FFTPACK	Fortran 1D FFT library by P. N. Swarztrauber [139].
Green	Code by John Green (v2.0, 1998). Only works for powers of 2.
GSL	C FFT routines from the GNU Scientific Library (GSL) version 0.3a. The FFT code was written by Brian Gough (1996).
Krukar	1D C FFT by R. H. Krukar.
Monnier	C FFT by Yves Monnier (1995).
Ooura	C and Fortran FFTs by Takuya Ooura (1996). They only work for sizes that are powers of 2. Includes real-complex and 2d transforms.
RMayer	C FFT by Ron Mayer (1993). Computes the DFT via the Hartley transform. Only works for powers of 2.
SCIPORT	Fortran FFT's from the SCIPORT package, a portable implementation of Cray's SCILIB library. These routines were developed at General Electric, probably by Scott H. Lamson. Only works for powers of 2, and includes real-complex routines. This code is an implementation of the Stockham auto-sort FFT algorithm.
Singleton	Mixed-radix, multidimensional, Fortran FFT by R. C. Singleton [132].
Sorensen	Fortran split-radix DIF FFT by H. V. Sorensen (1987). Includes real-complex transforms, and only works for powers of 2 [135].
***SUNPERF**	Sun Performance Library (UltraSPARC version 5.0)
Temperton	Fortran FFT in one and three dimensions by C. Temperton [142].

Table 6.1: Description of the programs benchmarked. All codes are generally available except for the entries marked with an asterisk, which are proprietary codes optimized for particular machines.

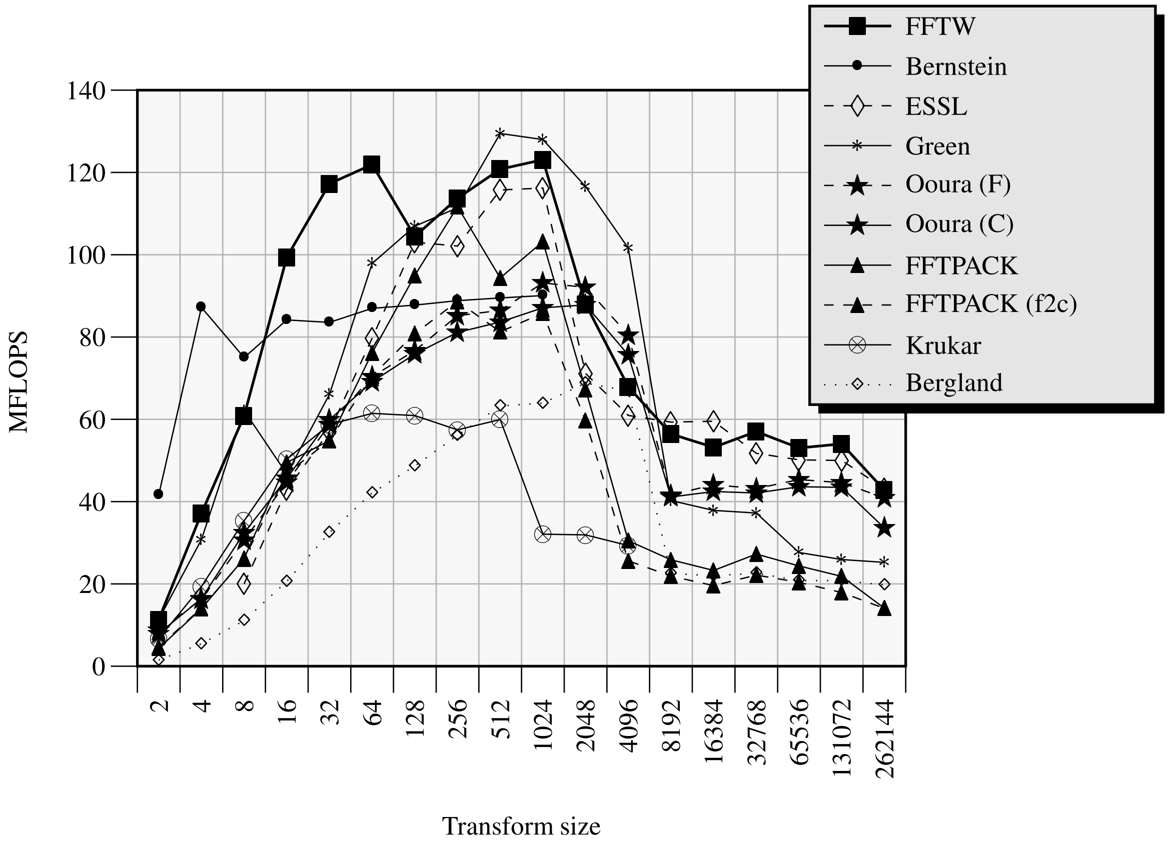


Figure 6-2: Comparison of complex FFTs for powers of 2 on RS/6000 Model 3BT (120-MHz POWER2). Compiled with `cc -O3 -qarch=pwrx -qtune=pwrx` and `f77 -O3 -qarch=pwr2 -qtune=pwr2`. AIX 3.2, IBM's `xlc` C compiler and `xlf90` Fortran compiler.

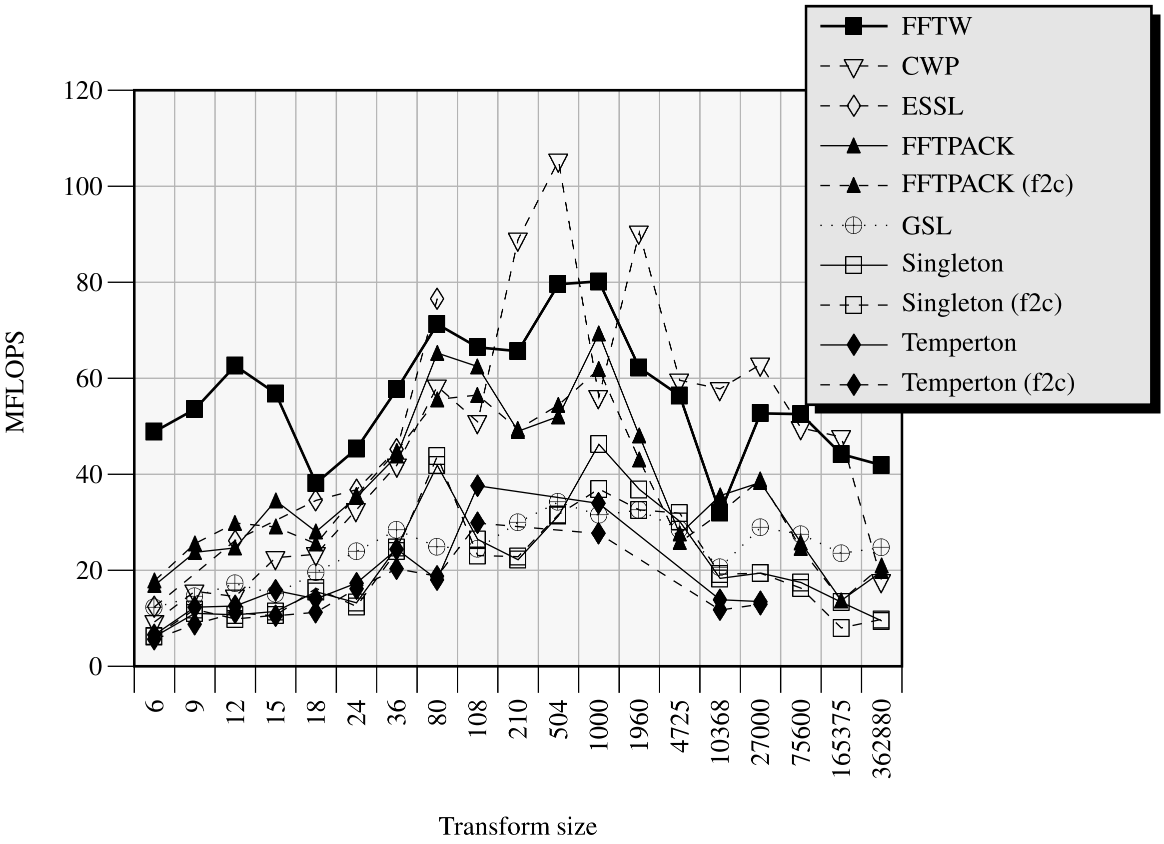


Figure 6-3: Comparison of complex FFTs for nonpowers of 2 on RS/6000 Model 3BT (120-MHz POWER2). See Figure 6-2 for the compiler flags.

(see [40, page 23] and [102, page 45]). For real transforms, we postulate that the transform requires $2.5n \lg n$ floating-point operations. Most FFT implementations (including FFTW) use algorithms with lower arithmetic complexity, and consequently the MFLOPS count is not an accurate measure of the processor performance. Although it is imprecise, this MFLOPS metric allows our numbers to be compared with other results in the literature [139], and it normalizes execution time so that we can display the results for different transform sizes on the same graph. All numbers refer to double precision transforms (64-bit IEEE floating point). Table 6.1 describes all FFT implementations for which we are showing performance results. Some codes in the benchmark are written in C, and others in Fortran; for some Fortran programs, we ran both the original code and a C translation produced by the free `f2c` software [51].

Figures 6-2 through 6-5 refer to the IBM RS/6000 Model 3BT machine. For powers of 2 (Figure 6-2), the strongest contenders are FFTW, IBM's ESSL library, and a program by John Green. FFTW is typically faster than ESSL, and it is faster than Green's code except for the range 512–4096. We shall see other cases where Green's program surpasses FFTW speed. The reason is

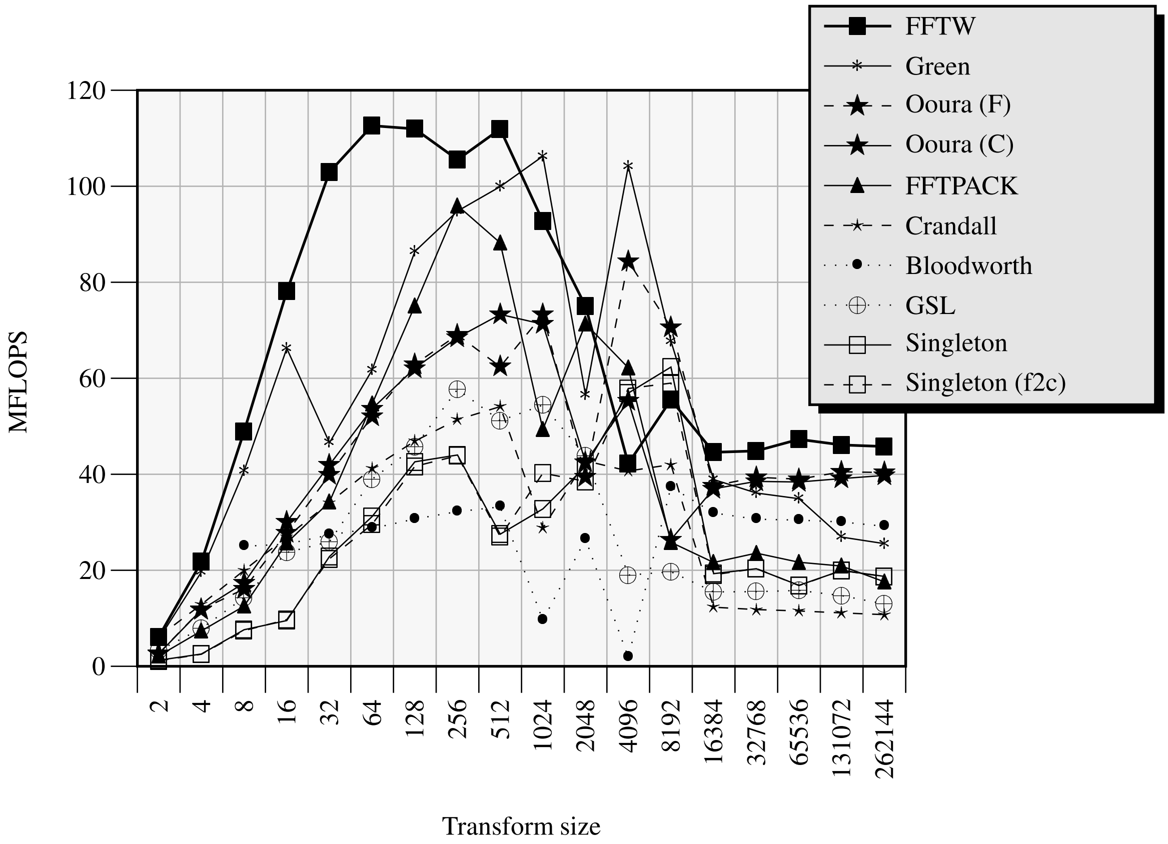


Figure 6-4: Comparison of real FFTs for powers of 2 on RS/6000 Model 3BT (120-MHz POWER2). See Figure 6-2 for the compiler flags.

that FFTW computes the transform out of place, i.e., with a separate input and output array, while Green's code computes the transform in place, and therefore FFTW uses twice as much memory as Green's program. For out-of-cache transforms, FFTW uses more memory bandwidth than Green's code. FFTW works out of place because no convenient in-place algorithm exists that works for general n. It is possible to implement a general in-place Fourier transform algorithm, but a complicated permutation is required to produce the proper output order. Green's program avoids this problem because it works only for powers of 2, where the permutation reduces to a simple bit-reversal. The program by Singleton [132] works in-place for many values of n, but it imposes seemingly inexplicable restrictions that derive from the implementation of the transposition. For example, if n has more than one square-free factor, the program requires that the product of the square-free factors be at most 210. Like the out-of-place library FFTPACK [139], FFTW opted for a consistent user interface to user's programs, even at the expense of performance.

Figure 6-3 shows complex transforms for nonpowers of 2. For these sizes, a remarkable program is the one labelled "CWP", which sometimes surpasses the speed of FFTW. The performance of CWP might not be directly comparable with that of other codes, because CWP is actually solving a different problem. Unlike all other programs we tried, CWP uses a prime-factor algorithm [140, 141] instead of the Cooley-Tukey FFT. The prime-factor algorithm works only when the size n of the transform can be factored into relatively prime integers (and therefore CWP does not work for powers of 2), but when it works, the prime-factor algorithm uses fewer operations than Cooley-Tukey. (FFTW currently does not implement the prime-factor algorithm at the executor level, although codelets do.) The CWP program only computes a transform of size n when n is the product of mutually prime factors from the set $\{2, 3, 4, 5, 7, 8, 9, 11, 13, 16\}$. You should be aware that some sizes displayed in the figure do not obey this restriction (for example, $1960 = 2^3 \cdot 5 \cdot 7^2$), in which case we ran CWP on a problem of the smallest acceptable size larger than the given size (like $1980 = 2^2 \cdot 3^2 \cdot 5 \cdot 11$). This is the normal *modus operandi* of the CWP library. A DFT of size n cannot simply be computed by padding the input with zeros and computing a DFT of larger size, however. It is possible to embedded a DFT into a DFT of larger size, using for example the "chirp" transform [121], but this embedding is nontrivial, and in any case, CWP does not perform any embedding. We included CWP in the benchmark because it uses interesting algorithms, and because it might be a viable choice in applications where one can choose the transform size.

Figure 6-4 shows results for real-to-complex transforms of size power of 2. Our previous remarks about Green's code apply here too. Figure 6-5 shows benchmark results for nonpowers of 2 real-to-complex transforms. We only had five codes available for this benchmark, since this kind of transform is particularly messy to program and only a handful implementations exist. (Luckily for us, in FFTW `genfft` produced all messy code automatically.)

The next set of figures (6-6 through 6-9) refer to a Sun HPC 5000 machine (167MHz UltraSPARC-I). For powers of 2 (Figure 6-6), FFTW succumbs to Sun's Performance Library in 4 cases out of 18, and it is slower than Green's program in 6 cases. For nonpowers of 2 (Figure 6-7), the fastest codes are FFTW, Sun's performance library, and CWP, where FFTW dominates for small sizes and the three codes are more or less in the same range for larger sizes. For real transforms, in the powers of 2 case (Figure 6-8) FFTW dominates everywhere except for 3 data points, and for other sizes (Figure 6-9) it is by far the fastest available code.

The third set of figures (6-10 through 6-13) refer to a DEC AlphaServer 4100 (467-MHz Alpha EV56). For powers of 2, complex data (Figure 6-10), we see a behavior similar to the IBM machine. FFTW is faster than all other codes for medium-sized transforms, but for large problems Green's program has again the advantage of a smaller memory footprint. For nonpowers of 2, complex data (Figure 6-11), CWP the fastest code for many big transforms—but recall that CWP is computing transforms of a different size which favors the algorithm that CWP uses. For real transforms (Figures 6-12 and 6-13) we see the familiar behavior where FFTW dominates in-cache transforms, but its performance drops below Green's for some big problems.

These figures show that for large transforms, FFTW is sometimes penalized because it is out-

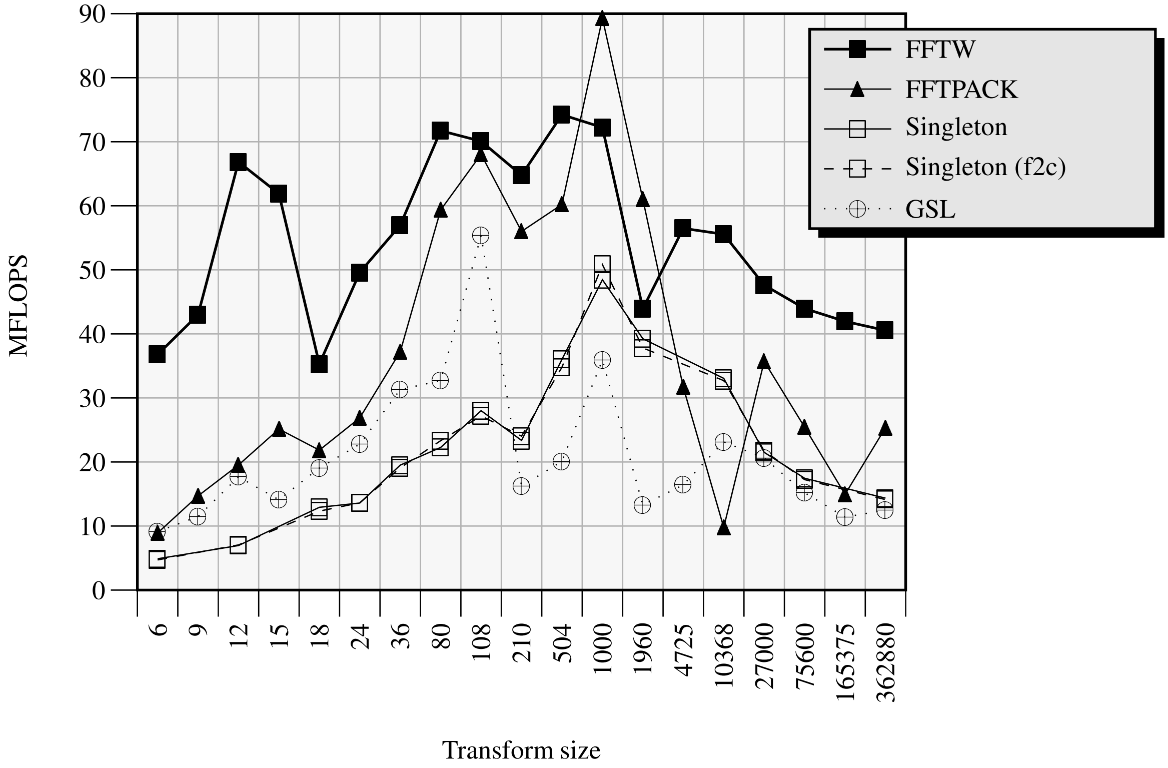


Figure 6-5: Comparison of real FFTs for nonpowers of 2 on RS/6000 Model 3BT (120-MHz POWER2). See Figure 6-2 for the compiler flags.

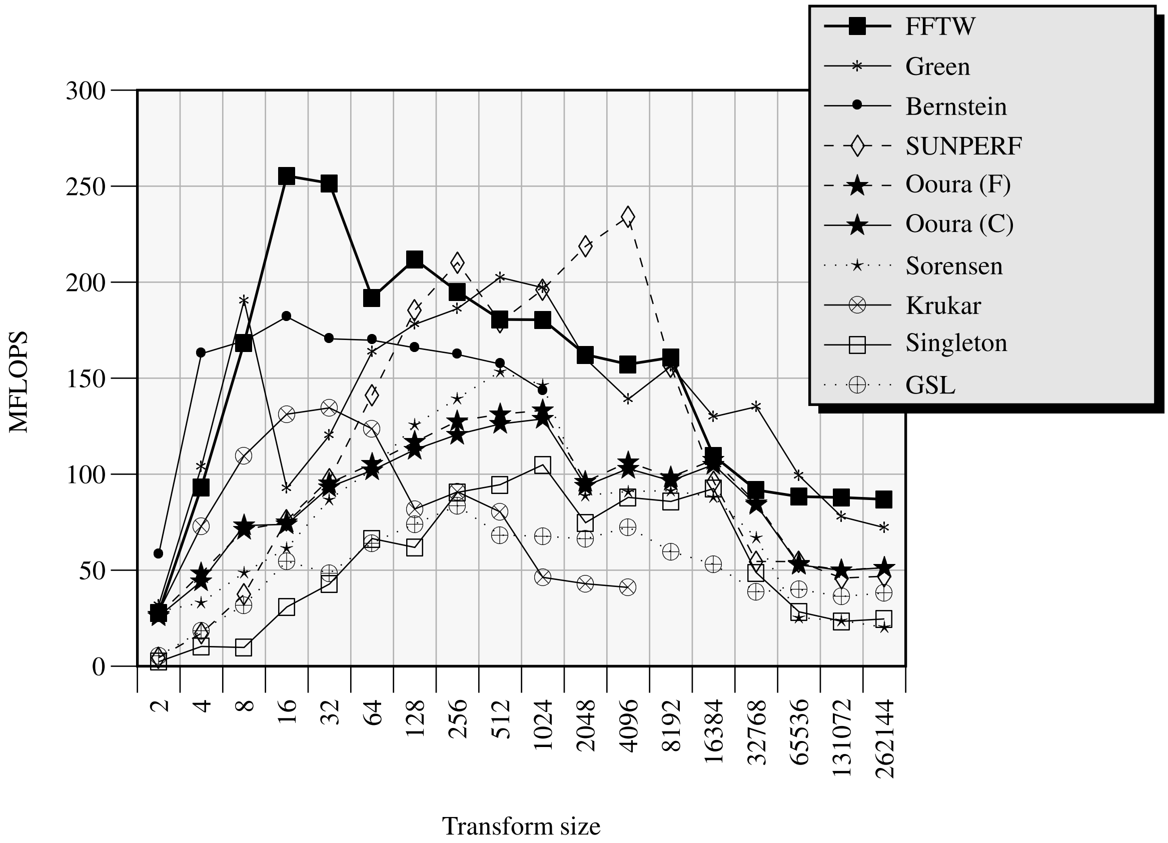


Figure 6-6: Comparison of complex FFTs for powers of 2 on a Sun HPC 5000 (167MHz UltraSPARC-I). Compiled with `cc -native -fast -xO5 -dalign -xarch=v9` and `f77 -fast -native -dalign -libmil -xO5 -xarch=v9`. SunOS 5.7, Sun WorkShop Compilers version 5.0.

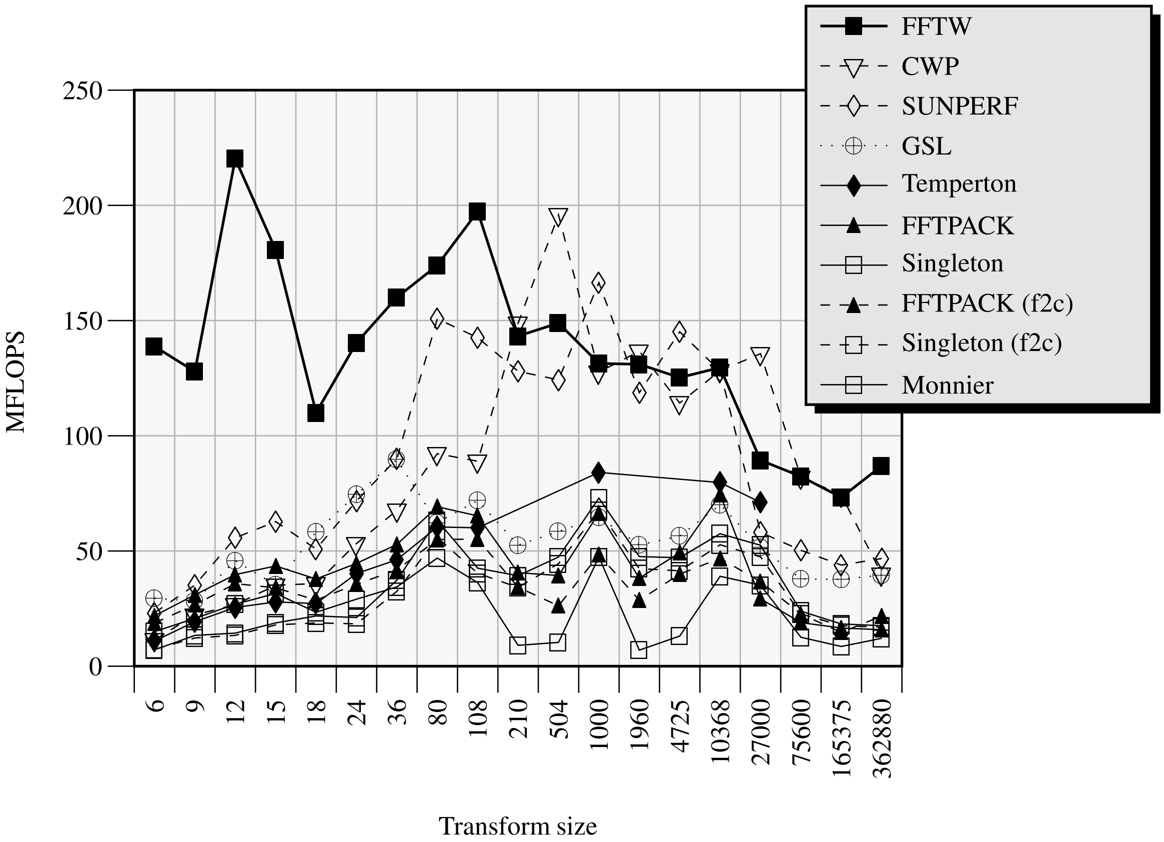


Figure 6-7: Comparison of complex FFTs for nonpowers of 2 on a Sun HPC 5000 (167MHz UltraSPARC-I). See Figure 6-6 for the compiler flags.

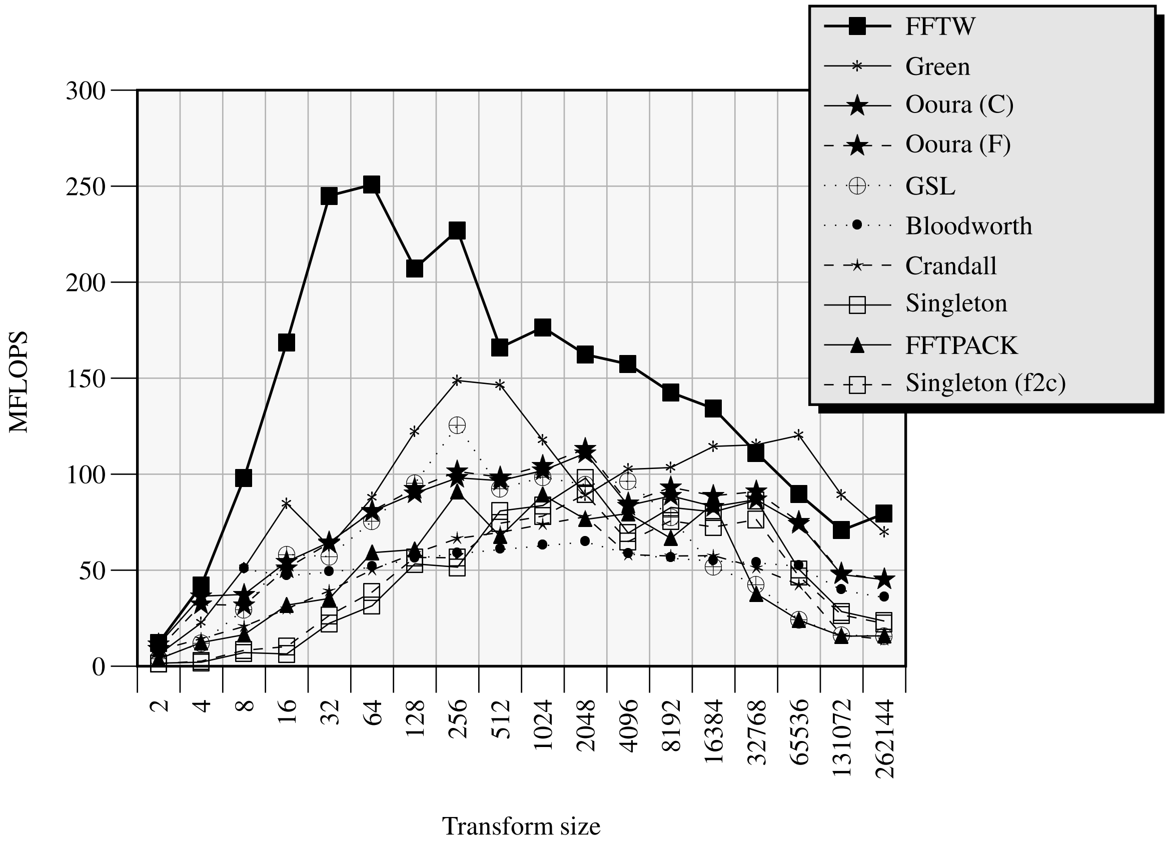


Figure 6-8: Comparison of real FFTs for powers of 2 on a Sun HPC 5000 (167MHz UltraSPARC-I). See Figure 6-6 for the compiler flags.

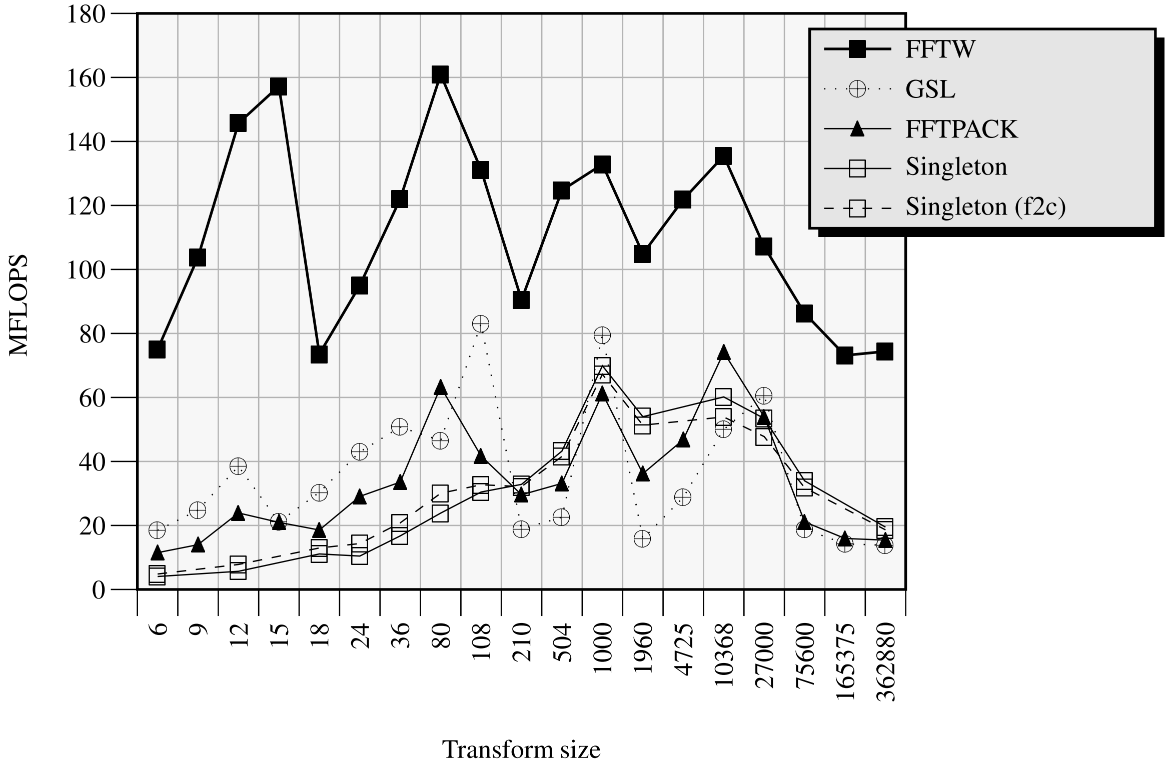


Figure 6-9: Comparison of real FFTs for nonpowers of 2 on a Sun HPC 5000 (167MHz UltraSPARC-I). See Figure 6-6 for the compiler flags.

of-place, a design choice dictated by our desire to build a general DFT library with a uniform user interface. For in-cache transforms, however, FFTW excels at extracting near-peak performance for in-cache transforms, showing that FFTW copes effectively with the intricacies of processor architectures as well or better than the best hand-tuned codes.

The results of a particular benchmark run were never entirely reproducible. Usually, the differences between runs of the same binary program were 5% or less, but small changes in the benchmark could produce much larger variations in performance, which proved to be very sensitive to the alignment of code and data in memory. We were able to produce changes of up to 30% in the benchmark results by playing with the data alignment (e.g. by adding small integers to the array sizes), or by changing the order in which different FFT routines were linked in the benchmark program. The numbers reported are not tweaked in any way, of course. The various FFT routines were linked in alphabetical order, and no special array alignment/padding was implemented.

6.3 FFTW's runtime structure

This section describes FFTW's runtime structure, which is comprised of the ***executor***—the part of FFTW that actually computes the transform—and the ***planner***, which implements FFTW's self-optimization capabilities. The planner uses a dynamic programming algorithm and runtime measurements to produce a fast composition of codelets. At the end of the section, we show that FFTW's planner is instrumental to attain portable high performance, since it can improve performance by a factor of 60% over a naive scheme that attempts to minimize the number of floating-point operations.

We start by describing the executor. The current release of FFTW employs several executors, for the various cases of complex, real-to-complex, and complex-to-real transforms, and for multiple dimensions. Here, we confine our discussion to the executor for complex one-dimensional transforms, which implements the Cooley-Tukey FFT algorithm [41] for transforms of composite size, and either Rader's algorithm or the definition Equation (6.1) for transforms of prime size.

With reference to Equation (6.3), the Cooley-Tukey algorithm centers around factoring the size N of the transform into $n = n_1 n_2$. The algorithm recursively computes n_2 transforms of size n_1, multiplies the results by certain constants traditionally called ***twiddle factors***, and finally computes n_1 transforms of size n_2. The executor consists of a C function that implements the algorithm just outlined, and of a library of ***codelets*** that implement special cases of the Cooley-Tukey algorithm. Specifically, codelets come in two flavors. ***Normal*** codelets compute the DFT of a fixed size, and are used as the base case for the recursion. ***Twiddle*** codelets are like normal codelets, but in addition they multiply their input by the twiddle factors. Twiddle codelets are used for the internal levels of the recursion. The current FFTW release contains codelets for all the integers up to 16 and all the powers of 2 up to 64, covering a wide spectrum of practical applications. Users who need transforms of special sizes (say, 19) can configure the executor for their needs by running `genfft` to produce specialized codelets.

The executor takes as input the array to be transformed, and also a ***plan***, which is a data structure that specifies the factorization of n as well as which codelets should be used. For example, here is a high-level description of a possible plan for a transform of length $n = 128$:

```
DIVIDE-AND-CONQUER(128, 4)
DIVIDE-AND-CONQUER(32, 8)
SOLVE(4)
```

In response to this plan, the executor initially computes 4 transforms of size 32 recursively, and then it uses the twiddle codelet of size 4 to combine the results of the subproblems. In the same way, the problems of size 32 are divided into 8 problems of size 4, which are solved directly using a normal codelet (as specified by the last line of the plan) and are then combined using a size-8 twiddle codelet.

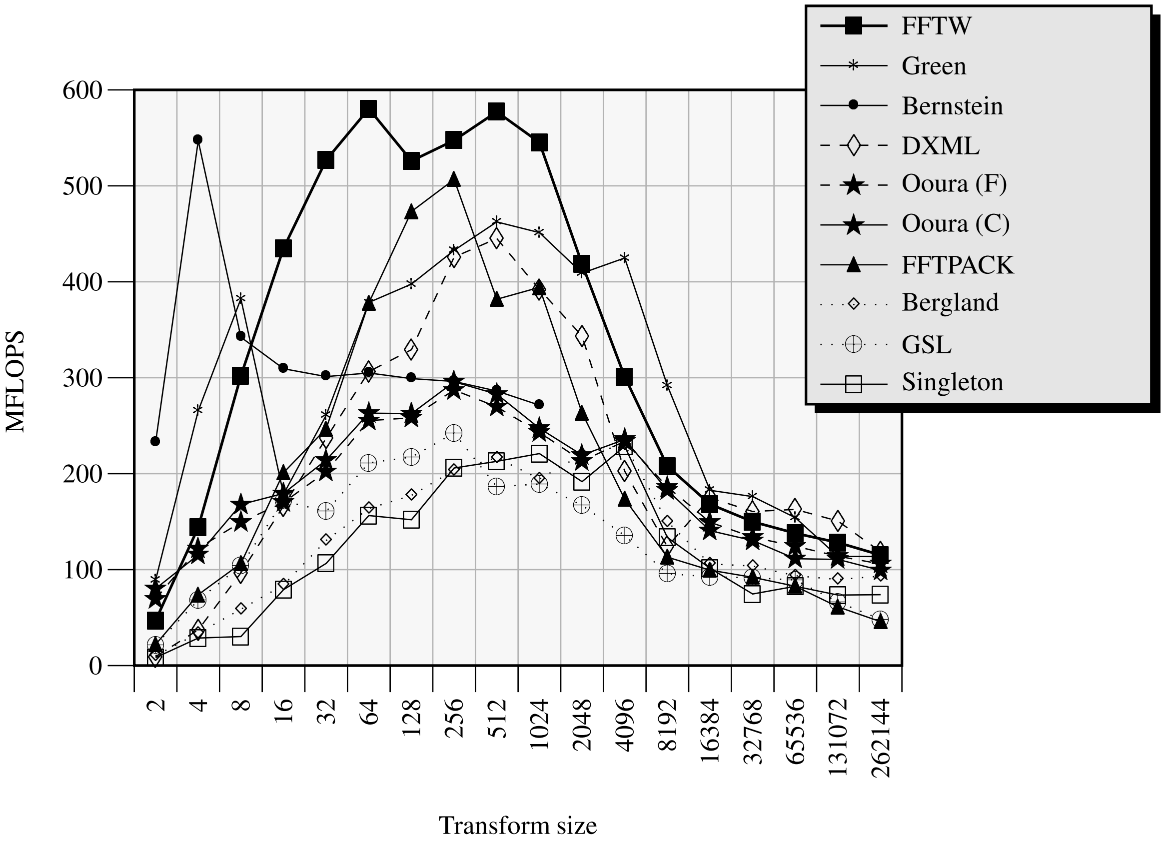


Figure 6-10: Comparison of complex FFTs for powers of 2 on a DEC AlphaServer 4100 (467-MHz Alpha EV56). Compiled with `cc -newc -w0 -O5 -ansi_alias -ansi_args -fp_reorder -tune host -arch host -std1` and `f77 -w0 -O5 -ansi_alias -ansi_args -fp_reorder -tune host -arch host -std1`. OSF1 V4.0, DEC C V5.6, DIGITAL Fortran 77 V5.1.

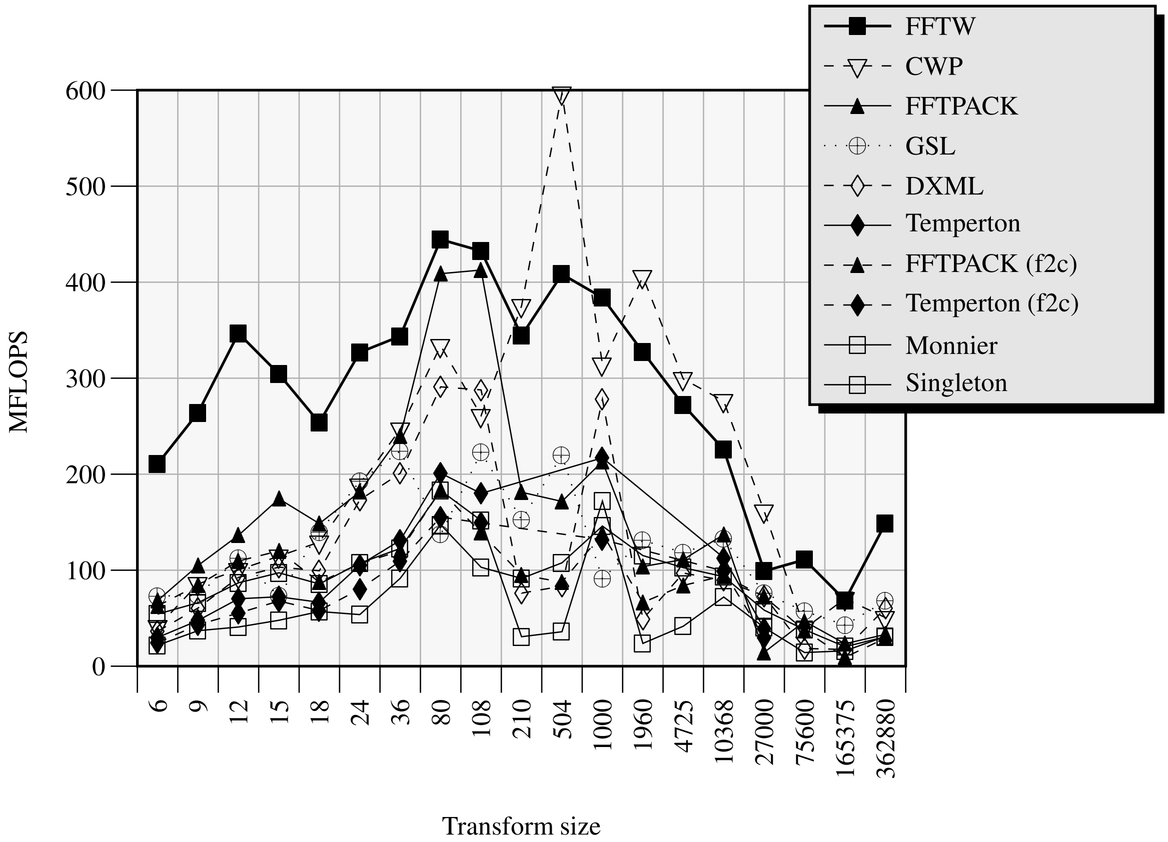


Figure 6-11: Comparison of complex FFTs for nonpowers of 2 on a DEC AlphaServer 4100 (467-MHz Alpha EV56). See Figure 6-10 for the compiler flags.

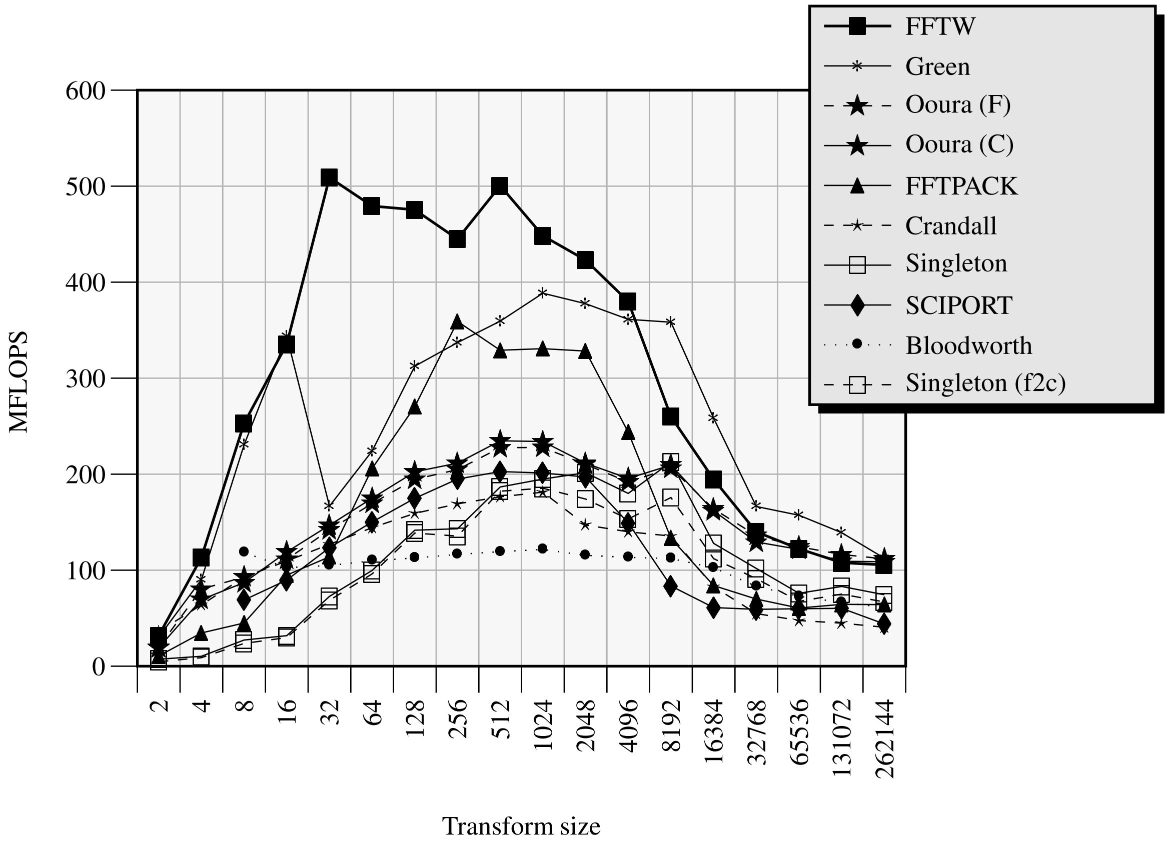


Figure 6-12: Comparison of real FFTs for powers of 2 on a DEC AlphaServer 4100 (467-MHz Alpha EV56). See Figure 6-10 for the compiler flags.

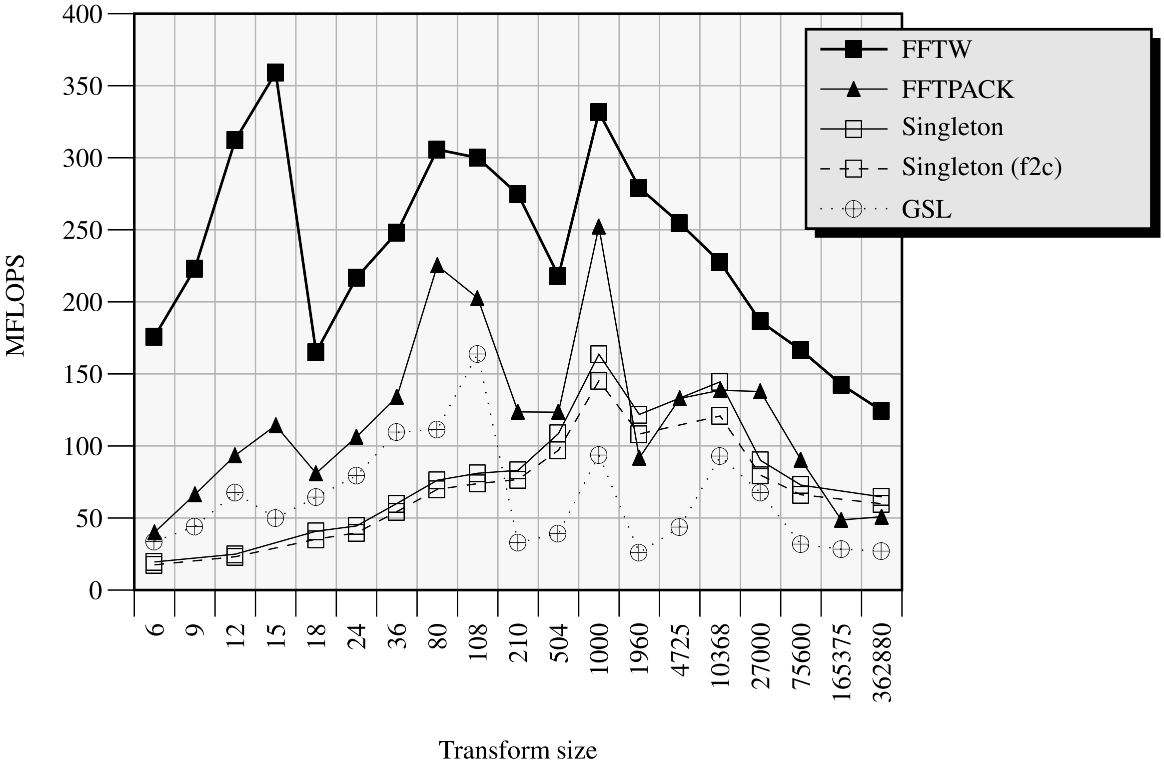


Figure 6-13: Comparison of real FFTs for nonpowers of 2 on a DEC AlphaServer 4100 (467-MHz Alpha EV56). See Figure 6-10 for the compiler flags.

The executor works by explicit recursion, in contrast with the traditional loop-based implementations [121, page 608]. This explicitly recursive implementation was motivated by considerations analogous to those discusses in Chapter 3: Divide and conquer is good for the memory hierarchy. As we saw in Chapter 3, as soon as a subproblem fits into the cache, no further cache misses are needed in order to solve that subproblem. Most FFT implementations benchmarked in Section 6.2 are loop based, and the benchmark results should convince you that divide and conquer does not introduce any unacceptable overhead. A precise evaluation of the relative merits of divide and conquer and loops would require the complete reimplementation of FFTW's planner and executor using loops, and the generation of a different set of codelets, and I have not yet performed this comparison.

Although we discussed an optimal cache-oblivious optimal FFT algorithm in Section 3.2, FFTW's executor does not implement it. Recall that the cache-oblivious algorithm works only for power-of-2 sizes, while FFTW is a general-purpose system that computes transforms of arbitrary size. Although the cache-oblivious algorithm can be generalized, the generalization involves a transposition that is tough to perform in the general case without using additional memory. I am investigating ways of implementing this algorithm efficiently, if only for powers of 2, since as we saw in Section 6.2, performance drops significantly as soon as the transform does not fit into cache.

How does one construct a good plan? FFTW's strategy is to ***measure*** the execution time of many plans and to select the best. This simple idea is one of the reasons of FFTW's high performance and portability. If a codelet happens to be fast on a given machine, for whatever reason, FFTW uses it. If the codelet is slow, FFTW does not use it. If the selection of codelets involves tradeoffs, the best tradeoff is found automatically.

Ideally, FFTW's ***planner*** should try all possible plans. This approach, however, is not practical due to the combinatorial explosion of the number of plans. Instead, the planner uses a dynamic-programming algorithm [42, chapter 16] to prune the search space. In order to use dynamic-programming, FFTW assumes ***optimal substructure***: if an optimal plan for a size n is known, this plan is still optimal when size n is used as a subproblem of a larger transform. This assumption is in principle false because of the different states of the cache and of the processor pipeline in the two cases. In practice, we tried both approaches and the simplifying hypothesis yielded good results, but the dynamic-programming algorithm runs much faster.

In order to demonstrate the importance of the planner, as well as the difficulty of predicting the optimal plan, in Figure 6-14 we show the speed of various plans (measured and reported as in Section 6.2) as a function of the number of floating point operations (flops) required by each plan. In this graph we can observe two important phenomena. First, different compositions of the codelets result in a wide range of performance, and it is important to choose the right combination. Second, the total number of flops is an inadequate predictor of the execution time, at least for the relatively small variations in the flops that obtain for a given n. As the figure shows, the fastest plan is about 60% faster than the one with the fewest operations.

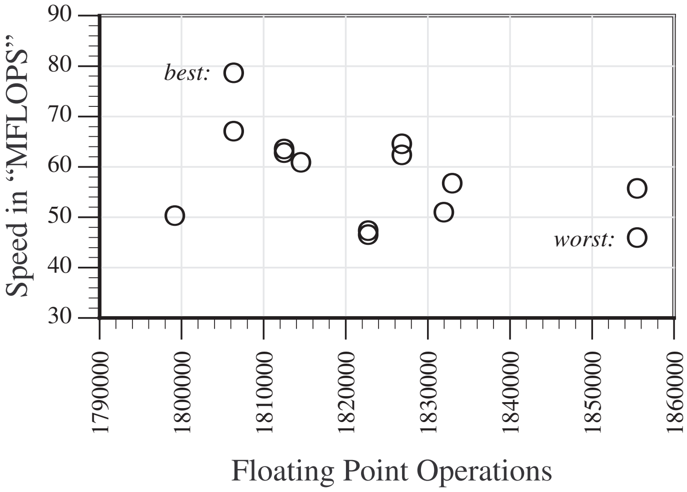


Figure 6-14: Speeds vs. flops of various plans considered by the planner for $n = 32768$. The "MFLOPS" unit of speed is described in Section 6.2. Notice that the fastest plan is not the one that performs the fewest operations. The machine is a Sun HPC 5000 (167MHz UltraSPARC-I). FFTW was compiled with `cc -native -fast -xO5 -dalign`. SunOS 5.5.1, Sun WorkShop Compilers version 4.2. (Note that the compiler is not the same as the one used in Figure 6-6. This test was performed with an older compiler.)

We have found that the optimal plan depends heavily on the processor, the memory architecture, and the compiler. For example, for double-precision complex transforms, $n = 1024$ is factored into $1024 = 8 \cdot 8 \cdot 16$ on an UltraSPARC and into $1024 = 32 \cdot 32$ on an Alpha. We currently have no theory that predicts the optimal plan, other than some heuristic rules of the form "codelet X seems to work best on machine Y."

6.4 The FFTW codelet generator

In this and in the following three sections, we focus our attention on `genfft`, the special-purpose compiler that generated 95% of FFTW's code. `genfft` shows the importance of ***metaprogramming*** in portable high-performance programs: instead of writing long sequences of optimized code by hand, it is easier to write a compiler that generates them. This section gives a high-level description of `genfft` and it explains how it is instrumental to achieve performance, portability, and correctness.

Codelets form the computational kernel of FFTW, but writing them by hand would be a long and error-prone process. Instead, FFTW's codelets are produced automatically by the ***FFTW codelet generator***, unimaginatively called `genfft`, which is an unusual special-purpose compiler. While a normal compiler accepts C code (say) and outputs numbers, `genfft` inputs the single integer n (the size of the transform) and outputs C code. `genfft` contains optimizations that are advantageous for DFT programs but not appropriate for a general compiler, and conversely, it does not contain optimizations that are not required for the DFT programs it generates (for example loop unrolling). It also contains optimizations that *are* appropriate both for a general-purpose compiler and for DFT programs, such as recursion unrolling, but that current compilers unfortunately do not implement.

As we have seen in Section 6.3, codelets come in two flavors: normal and twiddle. A normal codelet is just a fragment of C code that computes a Fourier transform of a fixed size (say, 16, or 19). For simplicity, we only focus on the generation of normal codelets, which compute Fourier transforms of a fixed size. Twiddle codelets are obtained by adding a multiplication stage to the inputs of a normal codelet.

`genfft`'s strategy is to express an FFT algorithm at a high level, and to automate all messy optimization details. As a consequence of this strategy, `genfft` operates in four phases.

1. In the ***dag creation*** phase, `genfft` produces a directed acyclic graph (dag) of the codelet, according to some well-known algorithm for the DFT, such as those from [48]. The generator contains many such algorithms and it applies the most appropriate. The algorithms used in this phase are almost literal translations of mathematical formulas such as Equation (6.1), without any optimization attempt.

2. In the ***simplifier***, `genfft` applies local rewriting rules to each node of the dag, in order to simplify it. This phase performs well-known algebraic transformations and common-subexpression elimination, but it also performs other transformations that are specific to the DFT. For example, it turns out that if all floating point constants are made positive, the generated code runs faster. (See Section 6.6.) Another important transformation is ***network transposition***, which derives from the theory of linear networks [44]. Moreover, besides noticing common subexpressions, the simplifier also attempts to create them. The simplifier is written in monadic style [151]. Using a monad, `genfft` deals with the dag as if it were a tree, which simplifies the implementation considerably.

3. In the ***scheduler***, `genfft` produces a cache-oblivious topological sort of the dag (a "schedule"), using the algorithm from Section 3.2. For transforms of size 2^k, this schedule implements the cache-oblivious algorithm from Section 3.2, and therefore it provably minimizes the asymptotic number of register spills, regardless of how many registers the target machine has. For transforms of other sizes the scheduling strategy is no longer provably good, but it still works well in practice. The scheduler depends heavily on the topological structure of DFT dags, and it would not be appropriate in a general-purpose compiler.

4. Finally, the schedule is unparsed to C. (It would be easy to produce FORTRAN or other languages by changing the unparser.) The unparser is rather obvious and uninteresting, except for one subtlety discussed in Section 6.9.

Although the creation phase uses algorithms that have been known for several years, the output of `genfft` is at times completely unexpected. For example, for a complex transform of size $n = 13$, the generator employs an algorithm due to Rader, in the form presented by Tolimieri and others

[144]. In its most sophisticated variant, this algorithm performs 172 real (floating-point) additions and 90 real multiplications. (See [103, Table VIII].) The generated code in FFTW for the same algorithm, however, contains 176 real additions and only 68 real multiplications. `genfft`'s output appears not to have been known before,[5] and it is among the best algorithms for this problem along with the algorithm from [131], which requires 188 additions and 40 multiplications. For reference purposes, Table 6.2 shows the operation counts of the DFT programs produced by `genfft`.

The generator specializes the dag automatically for the case where the input data are real, which occurs frequently in applications. This specialization is nontrivial, and in the past the design of an efficient real DFT algorithm required a serious effort that was well worth a publication [136]. `genfft`, however, automatically derives real DFT programs from the complex algorithms, and the resulting programs have the same arithmetic complexity as those discussed by [136, Table II].[6] The generator also produces real variants of the Rader's algorithm mentioned above, which to my knowledge do not appear anywhere in the literature.

`genfft` shows the important role of ***metaprogramming*** in portable high-performance programs. The philosophy of `genfft` is to separate the logic of an algorithm from its implementation. The user specifies an algorithm at a high level (the "program"), and also how he or she wants the code to be implemented (the "metaprogram"). Because of this structure, we achieve the following goals:

- *Performance* is the main goal of the FFTW project, and it could not have been achieved without `genfft`. For example, the codelet that performs a DFT of size 64 is used routinely by FFTW on the Alpha processor. As shown in Figure 6-10, this codelet is about 50% faster than any other code on that machine. The codelet consists of about 2400 lines of code, including 912 additions and 248 multiplications. Writing such a program by hand would be a formidable task for any programmer. At least for the DFT problem, these long sequences of straight-line code seem to be necessary in order to take full advantage of large CPU register sets and the scheduling capabilities of C compilers.

- *Portability* of FFTW's performance across diverse processor architectures is possible only because of `genfft`, because FFTW's self-optimizing machinery requires a large space of codelets in order to select the fast ones. Moreover, `genfft` enables portability to future systems. When next-generation microprocessors will be available with larger register sets and higher internal parallelism, even longer code sequences will be needed to exploit the new hardware fully. With `genfft`, it will be sufficient to ask the generator to produce larger codelets.

[5] In previous work [55], I erroneously claimed that `genfft`'s algorithm has the lowest known additive complexity for a DFT of size 13. I later discovered that in fact, the algorithm from [103] uses 4 fewer additions than `genfft`'s algorithm, although it requires 22 more multiplications.

[6] In fact, `genfft` saves a few operations in certain cases, such as $n = 15$.

	Complex		Real to complex		Complex to real	
size	adds	muls	adds	muls	adds	muls
2	4	0	2	0	2	0
3	12	4	4	2	4	2
4	16	0	6	0	6	2
5	32	12	12	6	12	7
6	36	8	14	4	14	4
7	60	36	24	18	24	19
8	52	4	20	2	20	6
9	80	40	38	26	32	18
10	84	24	34	12	34	14
11	140	100	60	50	60	51
12	96	16	38	8	38	10
13	176	68	76	34	76	35
14	148	72	62	36	62	38
15	156	56	64	25	64	31
16	144	24	58	12	58	18
17	296	116	116	58	116	63
18	196	80	102	60	82	36
19	428	228	276	174	272	175
20	208	48	86	24	86	30
21	264	136	112	63	112	71
22	324	200	142	100	142	102
23	692	484	284	244	284	247
24	252	44	104	20	104	30
25	352	184	204	140	152	98
26	404	136	178	68	178	70
27	380	220	237	169	164	102
28	352	144	150	72	150	78
29	760	396	300	202	300	207
30	372	112	162	56	158	52
31	804	340	320	162	322	167
32	372	84	156	42	156	54
64	912	248	394	124	394	146
128	2164	660	956	330	956	374

Table 6.2: Operation counts for complex, real-to-complex, and complex-to-real Fourier transform programs generated by `genfft`.

- Achieving *correctness* has been surprisingly easy. The DFT algorithms in `genfft` are encoded straightforwardly using a high-level language. The simplification phase transforms this high-level algorithm into optimized code by applying simple algebraic rules that are easy to verify. In the rare cases during development when the generator contained a bug, the output was completely incorrect, making the bug manifest.

- *Rapid turnaround* was essential to achieve the performance goals. Because `genfft` separates the *specification* of a DFT algorithm from its *implementation*, one can quickly experiment with optimizations and determine their effect experimentally. For example, the minus-sign propagation trick that we will describe in Section 6.6 could be implemented in only a few lines of code and tested within minutes.

- The generator is effective because it can apply *problem-specific* code improvements. For example, the scheduler is effective only for DFT dags, and it would perform poorly for other computations. Moreover, the simplifier performs certain improvements that depend on the DFT being a linear transformation.

- Finally, `genfft` derived some *new algorithms*, as in the example $n = 13$ discussed above. While this dissertation does not focus on these algorithms *per se*, they are of independent theoretical and practical interest.

In the next three sections, we describe the operation of `genfft`. Section 6.5 shows how `genfft` creates a dag for a codelet. Section 6.6 describes how `genfft` simplifies the dag. Section 6.7 describes `genfft`'s cache-oblivious scheduler.

6.5 Creation of the expression dag

This section describes how `genfft` creates an expression dag by evaluating a DFT algorithm symbolically. Consistently with the metaprogramming philosophy of separating the algorithm with the implementation, in `genfft` we express DFT algorithms at a high level, almost "straight out of the DSP book," without worrying about optimization. This section first describes the data type that encodes a codelet dag. Then, we show how the Cooley-Tukey algorithm (Equation (6.3)) translates verbatim into Caml code.

We start by defining the `node` data type, which encodes an arithmetic expression dag. Each dag node represents an operator, and the node's children represent the operands. This is the same representation as the one generally used in compilers [9, Section 5.2]. A node in the dag can have more than one "parent", in which case the node represents a common subexpression. The Objective Caml definition of `node` is given in Figure 6-15, and it is straightforward. A node is either a real number (encoded by the abstract data type `Number.number`), a load of an input variable, a store of

```
type node =
  | Num of Number.number
  | Load of Variable.variable
  | Store of Variable.variable * node
  | Plus of node list
  | Times of node * node
  | Uminus of node
```

Figure 6-15: Objective Caml code that defines the `node` data type, which encodes an expression dag.

an expression into an output node, the sum of the children nodes, the product of two nodes, or the sign negation of a node. For example, the expression $a - b$, where a and b are input variables, is represented by `Plus [Load` a`; Uminus (Load` b`)]`.

The structure `Number` maintains floating-point constants with arbitrarily high precision. FFTW currently computes all constants with 50 decimal digits of precision, so that a user can use the quadruple precision floating-point unit on a processor such as the UltraSPARC. `Number` is implemented on top of Objective Caml's arbitrary-precision rationals. If you wish, this is an extreme form of portability: If machines with 100-digits floating-point accuracy ever become available, FFTW is ready to run on them. The structure `Variable` encodes the input/output nodes of the dag, and the temporary variables of the generated C code. For the purposes of this dissertation, variables can be considered an abstract data type that is never used explicitly.

The `node` data type encodes expressions over real numbers, since the final C output contains only real expressions. For creating the expression dag of the codelet, however, it is convenient to express the algorithms in terms of complex numbers. The generator contains a structure called `Complex`, which implements complex expressions on top of the `node` data type, in a straightforward way.[7] The type `Complex.expr` (not shown) is essentially a pair of `node`s.

We now describe the function `fftgen`, which creates a dag for a DFT of size n. In the current implementation, `fftgen` uses one of the following algorithms.

- A split-radix algorithm [48], if n is a multiple of 4. Otherwise,
- A prime factor algorithm (as described in [121, page 619]), if n factors into $n_1 n_2$, where $n_i \neq 1$ and $\gcd(n_1, n_2) = 1$. Otherwise,
- The Cooley-Tukey FFT algorithm (Equation (6.3)) if n factors into $n_1 n_2$, where $n_i \neq 1$. Otherwise,

[7] One subtlety is that a complex multiplication by a constant can be implemented with either 4 real multiplications and 2 real additions, or 3 real multiplications and 3 real additions [92, Exercise 4.6.4-41]. The current generator uses the former algorithm, since the operation count of the dag is generally dominated by additions. On most CPUs, it is advantageous to move work from the floating-point adder to the multiplier.

```
let rec cooley_tukey n1 n2 input sign =
  let tmp1 j2 = fftgen n1
    (fun j1 -> input (j1 * n2 + j2)) sign in
  let tmp2 i1 j2 =
    exp n (sign * i1 * j2) @* tmp1 j2 i1 in
  let tmp3 i1 = fftgen n2 (tmp2 i1) sign
  in
   (fun i -> tmp3 (i mod n1) (i / n1))
```

Figure 6-16: Fragment of the FFTW codelet generator that implements the Cooley-Tukey FFT algorithm. The infix operator `@*` computes the complex product. The function `exp n k` computes the constant $\exp(2\pi k\sqrt{-1}/n)$.

- (n is a prime number) Rader's algorithm for transforms of prime length [126] if $n = 5$ or $n \geq 13$. Otherwise,
- Direct application of the definition of DFT (Equation (6.1)).

We now look at the operation of `fftgen` more closely. The function has type

```
fftgen : int -> (int -> Complex.expr) ->
         int -> (int -> Complex.expr)
```

The first argument to `fftgen` is the size `n` of the transform. The second argument is a function `input` with type `int -> Complex.expr`. The application `(input i)` returns a complex expression that contains the `i`-th input. The third argument `sign` is either 1 or -1, and it determines the direction of the transform.

Depending on the size n of the requested transform, `fftgen` dispatches one of the algorithms mentioned above. We now discuss how `genfft` implements the Cooley-Tukey FFT algorithm. The implementation of the other algorithms proceeds along similar lines.

Objective Caml code that implements the Cooley-Tukey algorithm can be found in Figure 6-16. In order to understand the code, recall Equation (6.3). This equation translates almost verbatim into Objective Caml. With reference to Figure 6-16, the function application `tmp1 j2` computes the inner sum of Equation (6.3) for a given value of j_2, and it returns a function of i_1. (`tmp1` is curried over i_1, and therefore i_1 does not appear explicitly in the definition.) Next, `(tmp1 j2 i1)` is multiplied by the twiddle factors, yielding `tmp2`, that is, the expression in square braces in Equation (6.3). Next, `tmp3` computes the outer summation, which is itself a DFT of size n_2. (Again, `tmp3` is a function of i_1 and i_2, curried over i_2.) In order to obtain the i-th element of the output of the transform, the index i is finally mapped into i_1 and i_2 and `(tmp3 i1 i2)` is returned.

Observe that the code in Figure 6-16 does not actually perform any computation. Instead, it builds a symbolic expression dag that specifies the computation. The other DFT algorithms are implemented in a similar fashion.

At the top level, the generator invokes `fftgen` with the size `n` and the direction `sign` specified by the user. The `input` function is set to `fun i -> Complex.load (Variable.input i)`, i.e., a function that loads the i-th input variable. Recall now that `fftgen` returns a function `output`, where `(output i)` is a complex expression that computes the i-th element of the output array. The top level builds a list of `Store` expressions that store `(output i)` into the i-th output variable, for all $0 \leq i < n$. This list of `Store`s is the codelet dag that forms the input of subsequent phases of the generator.

We conclude this section with a some remarks. According to the description given in this section, `fftgen` contains no special support for the case where the input is real. This statement is not completely true. In the actual implementation, `fftgen` maintains certain symmetries explicitly. For example, if the input is real, then the output is known to have hermitian symmetry. These additional constraints do not change the final output, but they speed up the generation process, since they avoid computing and simplifying the same expression twice. For the same reason, the actual implementation memoizes expressions such as `tmp1 i2 i1` in Figure 6-16, so that they are only computed once. These performance improvements were important for a user of FFTW who needed a hard-coded transform of size 101, and had not obtained an answer after the generator had run for three days. (See Section 6.9 for more details on the running time of `genfft`.)

At this stage, the generated dag contains many redundant computations, such as multiplications by 1 or 0, additions of 0, and so forth. `fftgen` makes no attempt to eliminate these redundancies. Figure 6-17 shows a possible C translation of a codelet dag at this stage of the generation process.

6.6 The simplifier

In this section, we present `genfft`'s ***simplifier***, which transforms code such as the one in Figure 6-17 into simpler code. This section is divided into two parts. We first discuss how the simplifier transforms the dag by applying algebraic transformations, common-subexpression elimination, minus-sign propagation and network transposition. Then, we discuss the actual implementation of the simplifier. Monads [151] form a convenient structuring mechanism for the code of the simplifier.

6.6.1 What the simplifier does

We begin by illustrating the improvements applied by the simplifier to a codelet dag. The simplifier traverses the dag bottom-up, and it applies a series of local improvements to every node. For explanation purposes, these improvements can be subdivided into three categories: algebraic transformations, common-subexpression elimination, and DFT-specific improvements. Since the first two kinds are well-known [9], I just discuss them briefly. We then consider the third kind in more detail.

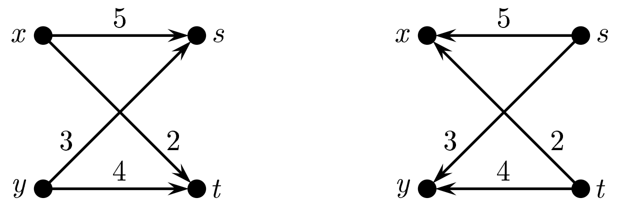
```
tmp1 = REAL(input[0]);
tmp5 = REAL(input[0]);
tmp6 = IMAG(input[0]);
tmp2 = IMAG(input[0]);
tmp3 = REAL(input[1]);
tmp7 = REAL(input[1]);
tmp8 = IMAG(input[1]);
tmp4 = IMAG(input[1]);
REAL(output[0]) = ((1 * tmp1) - (0 * tmp2))
      + ((1 * tmp3) - (0 * tmp4));
IMAG(output[0]) = ((1 * tmp2) + (0 * tmp1))
      + ((1 * tmp4) + (0 * tmp3));
REAL(output[1]) = ((1 * tmp5) - (0 * tmp6))
      + ((-1 * tmp7) - (0 * tmp8));
IMAG(output[1]) = ((1 * tmp6) + (0 * tmp5))
      + ((-1 * tmp8) + (0 * tmp7));
```

Figure 6-17: C translation of a dag for a complex DFT of size 2, as generated by `fftgen`. Variable declarations have been omitted from the figure. The code contains many common subexpression (e.g., `tmp1` and `tmp5`), and redundant multiplications by 0 or 1.

Algebraic transformations reduce the arithmetic complexity of the dag. Like a traditional compiler, the simplifier performs constant folding, and it simplifies multiplications by 0, 1, or -1, and additions of 0. Moreover, the simplifier applies the distributive property systematically. Expressions of the form $kx + ky$ are transformed into $k(x + y)$. In the same way, expressions of the form $k_1x + k_2x$ are transformed into $(k_1 + k_2)x$. In general, these two transformations have the potential of destroying common subexpressions, and they might increase the operation count. This does not appear to be the case for all DFT dags I have studied, although I do not fully understand the reason for this phenomenon.

Common-subexpression elimination is also applied systematically. Not only does the simplifier eliminate common subexpressions, it also attempts to create new ones. For example, it is common for a DFT dag (especially in the case of real input) to contain both $x-y$ and $y-x$ as subexpressions, for some x and y. The simplifier converts both expressions to either $x-y$ and $-(x-y)$, or $-(y-x)$ and $y-x$, depending on which expression is encountered first during the dag traversal.

The simplifier applies two kinds of ***DFT-specific improvements***. First, all numeric constants are made positive, possibly propagating a minus sign to other nodes of the dag. This curious transformation is effective because constants generally appear in pairs k and $-k$ in a DFT dag. To my knowledge, every C compiler would store both k and $-k$ in the program text, and it would load both constants into a register at runtime. Making all constants positive reduces the number of loads of constants by a factor of two, and this transformation alone speeds up the generated codelets by 10-15% on most machines. This transformation has the additional effect of converting subexpressions



$$\begin{pmatrix} s \\ t \end{pmatrix} = \begin{pmatrix} 5 & 3 \\ 2 & 4 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \qquad \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 5 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} s \\ t \end{pmatrix}$$

Figure 6-18: Illustration of "network" transposition. Each graph defines an algorithm for computing a linear function. These graphs are called ***linear networks***, and they can be interpreted as follows. Data are flowing in the network, from input nodes to output nodes. An edge multiplies data by some constant (possibly 1), and each node is understood to compute the sum of all incoming edges. In this example, the network on the left computes $s = 5x+3y$ and $t = 2x+4y$. The network on the right is the "transposed" form of the first network, obtained by reversing all edges. The new network computes the linear function $x = 5s + 2t$ and $y = 3s + 4t$. In general, if a network computes $x = My$ for some matrix M, the transposed network computes $y = M^T x$. (See [44] for a proof.) These linear networks are similar to but not the same as expression dags normally used in compilers and in `genfft`, because in the latter case the nodes and not the edges perform computation. A network can be easily transformed into an expression dag, however. The converse is not true in general, but it is true for DFT dags where all multiplications are by constants.

into a canonical form, which helps common-subexpression elimination.

The second DFT-specific improvement is not local to nodes, and is instead applied to the whole dag. The transformation is based on the fact that a dag computing a linear function can be "reversed" yielding a ***transposed*** dag [44]. This transposition process is well-known in the Signal Processing literature [121, page 309], and it operates a shown in Figure 6-18. It turns out that in certain cases the transposed dag exposes some simplifications that are not present in the original dag. (An example will be shown later.) Accordingly, the simplifier performs three passes over the dag. It first simplifies the original dag G yielding a dag G_1. Then, it simplifies the transposed dag G_1^T yielding a dag G_2^T. Finally, it simplifies G_2 (the transposed dag of G_2^T) yielding a dag G_3. (Although one might imagine iterating this process, three passes seem to be sufficient in all cases.) Figure 6-19 shows the savings in arithmetic complexity that derive from network transposition for codelets of various sizes. As it can be seen in the figure, transposition can reduce the number of multiplications, but it does not reduce the number of additions.

Figure 6-20 shows a simple case where transposition is beneficial. The network in the figure computes $c = 4 \cdot (2a + 3b)$. It is not safe to simplify this expression to $c = 8a + 12b$, since this transformation destroys the common subexpressions $2a$ and $3b$. (The transformation destroys 1 operation and 2 common subexpressions, which might increase the operation count by 1.) Indeed, the whole point of most FFT algorithms is to create common subexpressions. When the network is transposed, however, it computes $a = 2 \cdot 4c$ and $b = 3 \cdot 4c$. These transposed expressions *can* be safely transformed into $a = 8c$ and $b = 12c$ because each transformation saves 1 operation and

size	adds (not transposed)	muls (not transposed)	adds (transposed)	muls (transposed)
complex to complex				
5	32	16	32	12
10	84	32	84	24
13	176	88	176	68
15	156	68	156	56
real to complex				
5	12	8	12	6
10	34	16	34	12
13	76	44	76	34
15	64	31	64	25
complex to real				
5	12	9	12	7
9	32	20	32	18
10	34	18	34	14
12	38	14	38	10
13	76	43	76	35
15	64	37	64	31
16	58	22	58	18
32	156	62	156	54
64	394	166	394	146
128	956	414	956	374

Figure 6-19: Summary of the benefits of network transposition. The table shows the number of additions and multiplications for codelets of various size, with and without network transposition. Sizes for which the transposition has no effect are not reported in this table.

destroys 1 common subexpression. Consequently, the operation count cannot increase. In a sense, transposition provides a simple and elegant way to detect which dag nodes have more than one parent, which would be difficult to detect when the dag is being traversed.

6.6.2 Implementation of the simplifier

The simplifier is written in monadic style [151]. The monad performs two important functions: it allows the simplifier to treat the expression dag as if it were a tree, which makes the implementation considerably easier, and it performs common-subexpression elimination. We now discuss these two topics.

Treating dags as trees. Recall that the goal of the simplifier is to simplify an expression dag. The simplifier, however, is written as if it were simplifying an expression *tree*. The map from trees to dags is accomplished by memoization, which is performed implicitly by a monad. The monad maintains a table of all previously simplified dag nodes, along with their simplified versions. Whenever a node is visited for the second time, the monad returns the value in the table.

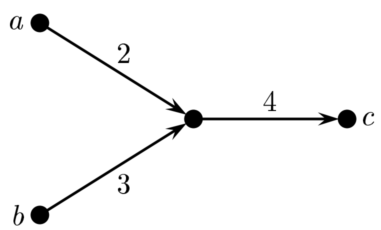


Figure 6-20: A linear network where which network transposition exposes some optimization possibilities. See the text for an explanation.

In order to fully understand this section, you really should be familiar with monads [151]. In any case, here is a very brief summary on monads. The idea of a monadic-style program is to convert all expressions of the form

```
let x = a in (b x)
```

into something that looks like

```
a >>= fun x -> returnM (b x)
```

The code should be read "call `f`, and then name the result `x` and return `(b x)`." The advantage of this transformation is that the meanings of "then" (the infix operator `>>=`) and "return" (the function `returnM`) can be defined so that they perform all sorts of interesting activities, such as carrying state around, perform I/O, act nondeterministically, etc. In the specific case of the FFTW simplifier, `>>=` is defined so as to keep track of a few tables used for memoization, and `returnM` performs common-subexpression elimination.

The core of the simplifier is the function `algsimpM`, as shown in Figure 6-21. `algsimpM` dispatches on the argument `x` (of type `node`), and it calls a simplifier function for the appropriate case. If the node has subnodes, the subnodes are simplified first. For example, suppose `x` is a `Times` node. Since a `Times` node has two subnodes `a` and `b`, the function `algsimpM` first calls itself recursively on `a`, yielding `a'`, and then on `b`, yielding `b'`. Then, `algsimpM` passes control to the function `stimesM`. If both `a'` and `b'` are constants, `stimesM` computes the product directly. In the same way, `stimesM` takes care of the case where either `a'` or `b'` is 0 or 1, and so on. The code for `stimesM` is shown in Figure 6-22.

Common-subexpression elimination (CSE) is performed behind the scenes by the monadic operator `returnM`. The CSE algorithm is essentially the classical bottom-up construction from [9, page 592]. The monad maintains a table of all nodes produced during the traversal of the dag. Each time a new node is constructed and returned, `returnM` checks whether the node appears elsewhere in the dag. If so, the new node is discarded and `returnM` returns the old node. (Two nodes are

```
let rec algsimpM x =
  memoizing
    (function
        Num a -> snumM a
      | Plus a ->
          mapM algsimpM a >>= splusM
      | Times (a, b) ->
          algsimpM a >>= fun a' ->
            algsimpM b >>= fun b' ->
              stimesM (a', b')
      | Uminus a ->
          algsimpM a >>= suminusM
      | Store (v, a) ->
          algsimpM a >>= fun a' ->
            returnM (Store (v, a'))
      | x -> returnM x)
    x
```

Figure 6-21: The top-level simplifier function `algsimpM`, written in monadic style. See the text for an explanation.

considered the same if they compute equivalent expressions. For example, $a + b$ is the same as $b + a$.)

The simplifier *interleaves* common-subexpression elimination with algebraic transformations. To see why interleaving is important, consider for example the expression $a - a'$, where a and a' are distinct nodes of the dag that compute the same subexpression. CSE rewrites the expression to $a - a$, which is then simplified to 0. This pattern occurs frequently in DFT dags.

The idea of using memoization for graph traversal is very old, but monadic style provides a particularly clean and modular implementation that isolates the memoization details. For example, the operator `>>=` in Figures 6-21 and 6-22 performs one step of common-subexpression elimination every time it is evaluated, it guarantees that `genfft` is not simplifying the same node twice, and so on. When writing the simplifier, however, we need not be concerned with this bookkeeping, and we can concentrate on the algebraic transformations that we want to implement.

6.7 The scheduler

In this section we discuss the `genfft` "cache-oblivious" scheduler, which produces a topological sort of the dag attempting to minimize register spills. For transforms whose size is a power of 2, `genfft` produces the cache-oblivious algorithm of Section 3.2, which is asymptotically optimal in terms of register usage even though the schedule is independent of the number of registers.

Even after simplification, a codelet dag of a large transform typically contains hundreds or even

```
let rec stimesM = function
  | (Uminus a, b) -> (* -a * b  ==> -(a * b) *)
      stimesM (a, b) >>= suminusM
  | (a, Uminus b) -> (* a * -b  ==> -(a * b) *)
      stimesM (a, b) >>= suminusM
  | (Num a, Num b) -> (* multiply two numbers *)
      snumM (Number.mul a b)
  | (Num a, Times (Num b, c)) ->
      snumM (Number.mul a b) >>= fun x ->
        stimesM (x, c)
  | (Num a, b) when Number.is_zero a ->
      snumM Number.zero   (* 0 * b  ==> 0 *)
  | (Num a, b) when Number.is_one a ->
      returnM b           (* 1 * b  ==> b *)
  | (Num a, b) when Number.is_mone a ->
      suminusM b          (* -1 * b  ==> -b *)
  | (a, (Num _ as b')) -> stimesM (b', a)
  | (a, b) -> returnM (Times (a, b))
```

Figure 6-22: Code for the function `stimesM`, which simplifies the product of two expressions. The comments (delimited with `(* *)`) briefly discuss the various simplifications. Even if it operates on a dag, this is exactly the code one would write to simplify a tree.

thousands of nodes, and there is no way to execute it fully within the register set of any existing processor. The scheduler attempts to reorder the dag in such a way that register allocators commonly used in compilers [115, Section 16] can minimize the number of register spills. Note that the FFTW codelet generator does not address the *instruction scheduling* problem; that is, the maximization of pipeline usage is left to the C compiler.

Figure 6-23 illustrates the scheduling problem. Suppose a processor has 5 registers, and consider a "column major" execution order that first executes all nodes in the shaded box (say, top-down), and then proceeds to the next column of nodes. Since there are 16 values to propagate from column to column, and the machine has 5 registers, at least 11 registers must be spilled if this strategy is adopted. A different strategy would be to execute all operations in the grey nodes before executing any other node. These operations can be performed fully within registers once the input nodes have been loaded. It is clear that different schedules lead to different behaviors with respect to register spills.

The problem of minimizing register spills is analogous to the problem of minimizing cache misses that we discusses in Chapter 3. The register set of a processor is a good approximation of an ideal cache with line size $L = 1$: Each memory location can be "cached" into any register (whence the register set is fully associative), and since a compiler knows the whole sequence of memory accesses in advance, it can implement the optimal replacement strategy by Belady [18]. (Although

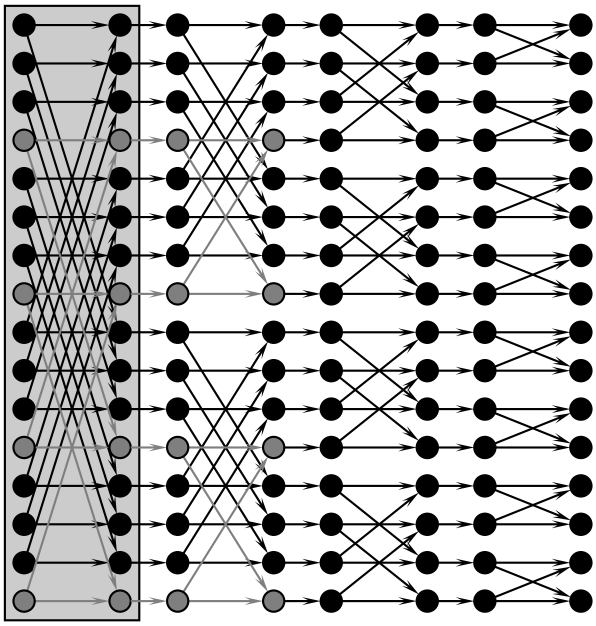


Figure 6-23: Illustration of the scheduling problem. The butterfly graph represents an abstraction of the data flow of the fast Fourier transform algorithm on 16 inputs. (In practice, the graph is more complicated because data are complex, and the real and imaginary part interact in nontrivial ways.) The shaded nodes and the shaded box denote two execution orders that are explained in the text.

this optimal strategy has been known for more than 30 years, real compilers might not employ it. See Section 6.9 for an example.)

To understand the operation of `genfft`'s scheduler, we now reexamine the cache-oblivious FFT algorithm from Section 3.2 in terms of the FFT dag like the one in Figure 6-23. Assume for now that n is a power of 2, because the cache-oblivious FFT algorithm only works in this case. The cache-oblivious algorithm partitions a problem of size n into $\sqrt{n}$ problems of size $\sqrt{n}$. This partition is equivalent to cutting the dag with a "vertical" line that partitions the dag into two halves of (roughly) equal size. (See Figure 6-24.) In the same way, `genfft` produces a schedule where every node in the first half is executed before any node in the second half. Each half consists of $\sqrt{n}$ connected components, which `genfft` schedules recursively in the same way in some arbitrary order.

The `genfft` scheduler uses this recursive partitioning technique for transforms of all sizes, not just powers of 2, although in general this partitioning is not provably cache-optimal, a lower bound on the cache complexity being unknown. Given any dag, the scheduler cuts the dag roughly into two halves. "Half a dag" is not well defined, however, except for the power of 2 case, and therefore the `genfft` scheduler uses a simple heuristic (described below) to compute the two halves for the general case. The cut induces a set of connected components that are scheduled recursively. The scheduler guarantees that all components in the first half of the dag (the one containing the inputs) are executed before the second half is scheduled.

Finally, we discuss the heuristic used to cut the dag into two halves. The heuristic consists of

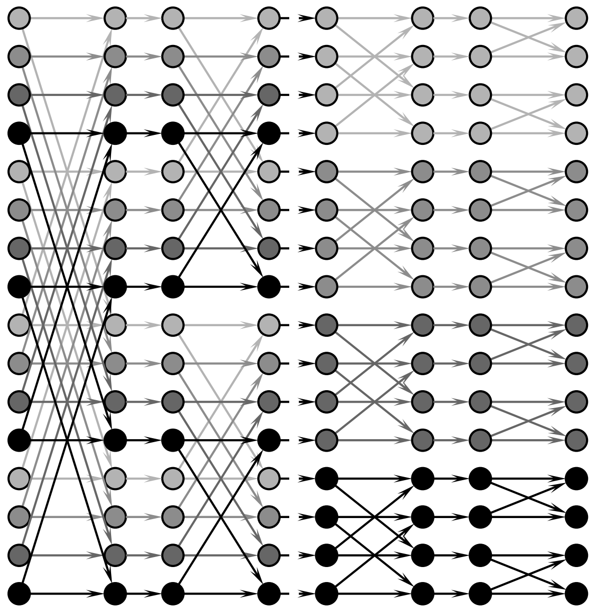


Figure 6-24: Illustration of the recursive partitioning operated by the `genfft` cache-oblivious scheduler. Like Figure 6-23, this figure shows the data flow dag of a FFT of 16 points. By cutting the dag in the "middle", as determined by the dashed lines, we produce $\sqrt{16} = 4$ connected components on each side of the cut. These components are shown in the figure with different shades of gray.

"burning the candle at both ends". Initially, the scheduler colors the input nodes red, the output nodes blue, and all other nodes black. After this initial step, the scheduler alternates between a red and a blue coloring phase. In a red phase, any node whose predecessors are all red becomes red. In a blue phase, all nodes whose successors are blue are colored blue. This alternation continues while black nodes exist. When coloring is done, red nodes form the first "half" of the dag, and blue nodes the second. When n is a power of 2, the FFT dag has a regular structure like the one shown in Figure 6-24, and this process has the effect of cutting the dag in the middle with a vertical line, yielding the desired optimal cache-oblivious behavior.

6.8 Real and multidimensional transforms

In this section, we discuss the implementation of real and multidimensional transforms in FFTW. Like complex transforms, the real transform code uses "normal" and "twiddle" codelets, and it employs its own planner and executor. The multidimensional code currently is built on top of one-dimensional transforms, that is, FFTW does not use multidimensional codelets.

Real one-dimensional transforms. FFTW computes real transforms using a planner and an executor similar to those of complex transforms. The executor currently implements a real variant of the Cooley-Tukey algorithm. Transforms of prime size are currently computed using Equation (6.1), and not by Rader's algorithm. Real input data occur frequently in applications, and a specialized real DFT code is important because the transform of a real array is an array with hermitian symmetry. Because of this symmetry, half of the output array is redundant and need not be computed and stored. Real transforms introduce two complications, however. First, hermitian arrays must be stored in such a way that the Cooley-Tukey recursion can be executed without performing complicated permutations. Second, the inverse transform can no longer be computed by conjugation of certain constants, because the input to the inverse transform is a hermitian array (as opposed to a real array) and the output is real (as opposed to hermitian).

FFTW stores a hermitian array $X[0 \dots n-1]$ into a real array $Y[0 \dots n-1]$ using the following ***halfcomplex*** storage layout. For all integers i such that $0 \leq i \leq \lfloor n/2 \rfloor$, we have $Y[i] = \text{Re}(X[i])$. For all integers i such that $0 < i < \lfloor n/2 \rfloor$, we have $Y[n-i] := \text{Im}(X[i])$. In other words, if $r_j = \text{Re}(X[j])$ and $i_j = \text{Im}(X[j])$, the array Y has the form:

$$r_0, r_1, r_2, \dots, r_{\lfloor n/2 \rfloor}, i_{\lfloor (n-1)/2 \rfloor}, \dots, i_2, i_1 .$$

This layout is a generalization of the layout presented in [136]. The name "halfcomplex" appears in the GNU Scientific Library (GSL)[59], which uses this layout for powers-of-2 transforms. This storage scheme is useful because n_1 halfcomplex arrays, each containing a transform of size n_2, can be combined in place to produce a transform of size $n_1 n_2$, just like in the complex case. This property is not true of layouts like the one used in FFTPACK [139], which stores a hermitian array by interleaving real and imaginary parts as follows.

Hence, the FFTW forward real executor is recursive and it contains two kinds of codelets. ***Real-to-halfcomplex*** codelets form the leaves of the recursion. Their input is a real array, and their output is the DFT of the input in halfcomplex order. ***Forward halfcomplex*** codelets combine small transforms (in halfcomplex order) to produce a larger transform. Similarly, the backward real executor uses ***halfcomplex-to-real*** codelets at the leaves of the recursion, and ***backward halfcomplex*** codelets in the intermediate stages. A backward halfcomplex codelet splits a large halfcomplex array into smaller arrays, that are then transformed recursively.

Multidimensional transforms. Multidimensional transforms are currently implemented on top of one-dimensional transforms. For example, a two-dimensional DFT of an array is computed by transforming all rows and then all columns (or vice versa). Alternatively, and more in the spirit of the rest of the FFTW system, we could use multidimensional codelets. For example, in the 2D case, we could employ two-dimensional codelets to "tile" the array. While it would be easy to modify `genfft` to produce the required codelets, this approach leads to an explosion in code size that is currently unacceptable, and the performance gains do not appear to justify the effort. This tradeoff will probably change once computers have so much memory that codelet size is not a problem. One drawback of the current implementation is that it is inefficient for small transforms. For example, on most processors it would be much faster to compute a 4×4 transform with a special codelet.

6.9 Pragmatic aspects of FFTW

This section discusses briefly the running time and the memory requirements of `genfft`, some problems that arise in the interaction of the `genfft` scheduler with C compilers, and FFTW's testing methodology.

Resource requirements. The FFTW codelet generator is not optimized for speed, since it is intended to be run only once. Indeed, users of FFTW can download a distribution of generated C code and never run `genfft` at all. Nevertheless, the resources needed by `genfft` are quite modest. Generation of C code for a transform of size 64 (the biggest used in FFTW) takes about 75 seconds on a 200MHz Pentium Pro running Linux 2.2 and the native-code compiler of Objective Caml 2.01. `genfft` needs less than 3 MB of memory to complete the generation. The resulting codelet contains 912 additions, 248 multiplications. On the same machine, the whole FFTW system can be regenerated in about 15 minutes. The system contains about 55,000 lines of code in 120 files, consisting of various kinds of codelets for forward, backward, real to complex, and complex to real transforms. The sizes of these transforms in the standard FFTW distribution include all integers up to 16 and all powers of 2 up to 64.

A few FFTW users needed fast hard-coded transforms of uncommon sizes (such as 19 and 23), and they were able to run the generator to produce a system tailored to their needs. The biggest program generated so far was for a complex transform of size 101, which required slightly less than two hours of CPU time on the Pentium Pro machine, and about 10 MB of memory. Again, a user had a special need for such a transform, which would be formidable to code by hand. In order to achieve this running time, I was forced to replace a linked-list implementation of associative tables by hashing, and to avoid generating "obvious" common subexpressions more than once when the dag is created. The naive generator was somewhat more elegant, but had not produced an answer after three days.

Interaction with C compilers. The long sequences of straight-line code produced by `genfft` can push C compilers (in particular, register allocators) to their limits. The combined effect of `genfft` and of the C compiler can lead to performance problems. The following discussion presents two particular cases that I found particularly surprising, and is not intended to blame any particular compiler or vendor.

The optimizer of the `egcs-1.1.1` compiler performs an instruction scheduling pass, followed by register allocation, followed by another instruction scheduling pass. On some architectures, including the SPARC and PowerPC processors, `egcs` employs the so-called "Haifa scheduler",

```
void foo(void)
{
 double a;
 double b;

 .. lifetime of a ..
 .. lifetime of b ..
}
```

```
void foo(void)
{
 {
  double a;
   .. lifetime of a ..
 }
 {
  double b;
   .. lifetime of b ..
 }
}
```

Figure 6-25: Two possible declarations of local variables in C. On the left side, variables are declared in the topmost lexical scope. On the right side, variables are declared in a private lexical scope that encompasses the lifetime of the variable.

which usually produces better code than the normal `egcs/gcc` scheduler. The first pass of the Haifa scheduler, however, has the unfortunate effect of destroying `genfft`'s schedule (computed as explained in Section 6.7). In `egcs`, the first instruction scheduling pass can be disabled with the option `-fno-schedule-insns`, and on a 167-MHz UltraSPARC I, the compiled code is between 50% and 100% faster and about half the size when this option is used. Inspection of the assembly code produced by `egcs` reveals that the difference consists entirely of register spills and reloads.

Digital's C compiler for Alpha (DEC C V5.6-071 on Digital UNIX V4.0 (Rev. 878)) seems to be particularly sensitive to the way local variables are declared. For example, Figure 6-25 illustrates two ways to declare temporary variables in a C program. Let's call them the "left" and the "right" style. `genfft` can be programmed to produce code in either way, and for most compilers I have tried there is no appreciable performance difference between the two styles. Digital's C compiler, however, appears to produce better code with the right style (the right side of Figure 6-25). For a transform of size 64, for example, and compiler flags `-newc -w0 -O5 -ansi_alias -ansi_args -fp_reorder -tune host -std1`, a 467MHz Alpha achieves about 450 MFLOPS with the left style, and 600 MFLOPS with the right style. (Different sizes lead to similar results.) I could not determine the exact source of this difference.

Testing FFTW. FFTW uses different plans on each platform, and some codelets are not used at all on the machines available to me. How do we ensure that FFTW is correct? FFTW uses the ***self-testing*** algorithm by Funda Ergün [49], a randomized test that guarantees that a given program computes the DFT for an overwhelmingly large fraction of all possible inputs. The self-tester does not require any other DFT program to be available. In the past, we checked FFTW against the program by Singleton [132], assuming that any bug in the program would have been found in the thirty years passed since the program was written. Unfortunately, while Singleton's routine is correct, one of the FORTRAN compilers we used was not. Besides, Singleton's program does not work for all input sizes, while FFTW does, and thus we could not test FFTW fully. In contrast, Ergün's tester is fast, easy to code, and it works for all sizes. Computer theoreticians have developed many testing techniques that possess similar advantages, but regrettably, these techniques seem to be mostly unknown to practitioners. I definitely recommend that any programmer become familiar with this beautiful topic; see [24] for a gentle introduction.

6.10 Related work

Other systems exist with self-optimization capabilities. PHiPAC [22] generates automatically-tuned matrix-multiplication kernels by generating many C programs and selecting the fastest. In most cases, PHiPAC is able to beat hand-optimized BLAS routines. PHiPAC predates FFTW [21], but I became acquainted with it only after the publication of [22] in July 1997, after the release of FFTW-1.0 in March 1997. PhiPAC and FFTW focus on complementary aspects of self-optimization. PHiPAC automatically optimizes the multiplication kernels, which correspond to FFTW's codelets, while FFTW optimizes compositions of codelets, or plans, and it relies on `genfft` to produce good codelets. Consequently, FFTW's self-optimization occurs at runtime, while PHiPAC operates at installation time and it is not needed after the kernels have been generated. Because of the mathematical richness of the Fourier transform, FFTW employs a sophisticated compiler that focuses on algebraic transformations and on cache-oblivious scheduling. On the other hand, PHiPAC uses the standard matrix multiplication algorithm, and it is concerned with scheduling it appropriately for a processor's pipeline. Both approaches are legitimate and effective techniques for portable high performance, and I expect FFTW to evolve to produce codelets tailored to a single machine, in the same spirit of PHiPAC.

The Linux kernel included in Redhat 6.0 incorporates many routines that compute checksums in the RAID disk drivers. At boot time, the kernel measures the execution time of the various subroutines and uses the fastest.

Researchers have been generating FFT programs for at least twenty years, possibly to avoid the tedium of getting all the implementation details right by hand. To my knowledge, the first generator of FFT programs was FOURGEN, written by J. A. Maruhn [108]. It was written in PL/I and it generated FORTRAN.[8] FOURGEN is limited to transforms of size 2^k.

Perez and Takaoka [123] present a generator of Pascal programs implementing a prime factor

[8]Maruhn argues that PL/I is more suited than FORTRAN to this program-generation task, and has the following curious remark:

> One peculiar difficulty is that some FORTRAN systems produce an output format for floating-point numbers without the exponent delimiter "E", and this makes them illegal in FORTRAN statements.

FFT algorithm. This program is limited to complex transforms of size n, where n must be factorable into mutually prime factors in the set $\{2, 3, 4, 5, 7, 8, 9, 16\}$.

Johnson[9] and Burrus [86] applied dynamic programming to the automatic design of DFT modules. Selesnick and Burrus [131] used a program to generate MATLAB subroutines for DFT's of certain prime sizes. In many cases, these subroutines are the best known in terms of arithmetic complexity.

The EXTENT system by Gupta and others [74] generates FORTRAN code in response to an input expressed in a ***tensor product*** language. Using the tensor product abstraction one can express concisely a variety of algorithms that includes the FFT and matrix multiplication (including Strassen's algorithm).

Another program called `genfft` generating Haskell FFT subroutines is part of the `nofib` benchmark for Haskell [122]. Unlike my program, this `genfft` is limited to transforms of size 2^k. The program in `nofib` is not documented at all, but apparently it can be traced back to [77].

Veldhuizen [146] used a template metaprograms technique to generate `C++` programs. The technique exploits the template facility of `C++` to force the `C++` compiler to perform computations at compile time.

All these code generators are restricted to complex transforms, and the FFT algorithm is known *a priori*. To my knowledge, the FFTW generator is the only one that produces real algorithms, and in fact, which can *derive* real algorithms by specializing a complex algorithm. Also, my generator is the only one that addressed the problem of scheduling the program efficiently.

6.11 Conclusion

Current computer systems are so complex that their behavior is unpredictable. Ironically, while performance is the very reason for this complexity, peak performance is almost impossible to attain because of lack of predictability. Only time will tell whether we will regret having designed machines so complex. In the meanwhile, in this chapter we showed that a software system that is aware of its own performance can achieve high performance with no tuning. For the case of FFTW, a special-purpose compiler is a necessary component of such a self-optimizing system, because we need a sufficiently large space of algorithmic variations to be able to pick the most effective.

From another point of view, this chapter presented a real-world application of domain-specific compilers and of advanced programming techniques, such as monads. In this respect, the FFTW experience has been very successful: the current release FFTW-2.1.2 is being downloaded by more than 100 people every week, and a few users have been motivated to learn ML after their experience with FFTW. In the rest of this concluding section, I offer some ideas about future work and possible

[9] Unrelated to Steven G. Johnson, the other author of FFTW.

developments of the FFTW system.

The current `genfft` program is somewhat specialized to computing linear functions, using algorithms whose control structure is independent of the input. Even with this restriction, the field of applicability of `genfft` is potentially huge. For example, signal processing FIR and IIR filters fall into this category, as well as other kinds of transforms used in image processing (for example, the discrete cosine transform used in JPEG). I am confident that the techniques described in this chapter will prove valuable in this sort of application.

Recently, I modified `genfft` to generate crystallographic Fourier transforms [12]. In this particular application, the input consists of 2D or 3D data with certain symmetries. For example, the input data set might be invariant with respect to rotations of 60 degrees, and it is desirable to have a special-purpose FFT algorithm that does not execute redundant computations. Preliminary investigation shows that `genfft` is able to exploit most symmetries. I am currently working on this problem.

In its present form, `genfft` is somewhat unsatisfactory because it intermixes programming and metaprogramming. At the programming level, one specifies a DFT algorithm, as in Figure 6-16. At the metaprogramming level, one specifies how the program should be simplified and scheduled. In the current implementation, the two levels are confused together in a single binary program. It would be nice to build a general-purpose "metacompiler" that clearly separates programming from metaprogramming and allows other problems to be addressed in a similar fashion.

Chapter 7

Conclusion

[T]here ain't nothing more to write about, and I am rotten glad of it, because if I'd a knowed what a trouble it was to make a book I wouldn't a tackled it and aint't agoing to no more.

(Huckleberry Finn)

In this concluding chapter, we look at some ideas for future work, and we finally summarize the main ideas of this thesis.

7.1 Future work

Portable high-performance I/O. The topic of portable high-performance disk I/O was not addressed at all in this document. We can identify two general research topics in this area, roughly inspired by cache-oblivious algorithms and Cilk. The first topic is to design "disk-geometry-oblivious" data structures for single (i.e., not parallel) disks. The second topic is to extend the Cilk model with provably efficient parallel I/O.

Disk access time depends on the geometrical and mechanical properties of disks. Current disks are partitioned into *cylinders*, and cylinders are divided into *sectors*. Data within the same sector can be accessed quickly with one operation. Accesses within the same cylinder are slower than accesses within a sector, but faster than accesses to another cylinder. In this latter case, the speed of intra-cylinder accesses depends on the physical distance between the old and the new cylinder. With current technology, the number of sectors per cylinder is not constant, since cylinders in the outer part of the disk comprise a larger area and thus can host more sectors.

It should be possible to design "cache-oblivious" data structures to store data on a disk. Suppose for example that we want to store a binary search tree on a disk. If a disk "cache line" (the unit of transfer between disk and memory, usually called a *page* or a *block*) contains L elements, it is a good idea to group subtrees of height $\lg L$, as explained in [91], so that a tree of n elements can be searched in $\log_L n$ page accesses. This disk-aware layout depends on L, but it is possible to devise a "disk-oblivious" tree layout by cutting the tree at level $(\lg n)/2$ and storing the resulting $\Theta(\sqrt{n})$ subtrees in a recursive fashion. This "disk-oblivious" layout has the same asymptotic I/O complexity as the disk-aware one. I conjecture that this layout is insensitive to the variable number of sectors per cylinder; if true, this conjecture would show a nice advantage of cache-oblivious algorithms over cache-aware ones. The ideal-cache theory does not model the intra-cylinder physical distance, however. Is there a "disk-oblivious" way to store a binary tree on disk so as to minimize the total execution time of the search, no matter what the parameters of the disk are? Indeed, the whole topic of cache- and disk-oblivious data structures has not been investigated yet, and I would expect such an investigation to yield useful algorithms and programming paradigms. For example, can we design a cache/disk-oblivious B-tree?

Concerning parallel I/O, it would be nice to extend the Cilk system with I/O in a way that preserves Cilk's performance guarantees. Since files can be used to simulate shared memory, I expect the solution to this problem to depend on the consistency model that we use for files. Location consistency and other traditional memory models seem inadequate for the case of files, however. For example, the "parallel append" file operation appears to be useful. In a parallel append, a file is opened and two parallel threads are spawned to append data to the file. The output is the same as if the C elision of the Cilk program had been executed, regardless of how many processors execute the parallel program. How to implement parallel append preserving the performance of the Cilk scheduler is an open problem.

Extensions to Cilk. The Cilk system needs to be extended to support other kinds of synchronization, such as producer-consumer relationships and mutual exclusion. Currently, the Cilk-5 implementation of locks is an afterthought that invalidates many of Cilk's performance guarantees. Even worse, there is no linguistic support for locks in Cilk (the Nondeterminator will detect data races in programs that use locks [37], however). How to incorporate easy-to-use and efficient synchronization in a general-purpose programming language is a tough problem that nobody has fully solved yet. If you find a solution, submit it immediately to the Java and Perl authors before the World-Wide Web collapses because of incorrect protocols.

From the point of view of the Cilk implementation, Cilk needs work in two directions. First, Cilk for SMP's should be made easily available to the general public. Although every version of Cilk has been publicly released, and although Cilk-5 is relatively bug-free and robust, the system is still a research prototype. Cilk is mature enough to become a "product," and it is time to write a production-quality system, which should be distributed with Linux and other operating systems so that many people can use it. Second, Cilk needs to be implemented on distributed-memory systems such as networks of workstations. An implementation was written by Keith Randall [127] for Unix systems, but this implementation is still preliminary. The main problem is the implementation of shared memory, using BACKER or its variants described in [127]. Keith's implementation uses the Unix user-level virtual-memory system, but this solution is too slow. It seems necessary to implement BACKER in the Unix kernel, where it can use the virtual-memory and network subsystems without too many overheads. Fortunately, the Linux kernel is currently robust and mature enough that such an implementation is feasible and will probably be efficient.

Extensions to FFTW. The current FFTW system covers most of the spectrum of practical uses of Fourier transforms, but it would be nice to extend it to compute related transforms, such as the discrete cosine transform (DCT) and maybe the Hartley transform [31]. Currently, `genfft` is capable of generating DCT programs, but the planner/executor machinery has not been implemented.

We should implement a planner for multidimensional transforms and an executor that uses multidimensional codelets. I expect performance improvements at least for small transforms (say, $4 \times 4 \times 4$ or 8×8), which can be unrolled as straight-line code. The 8×8 DCT is especially important because it is used in the JPEG image compression standard.

Open problems in cache-obliviousness. The limits of cache obliviousness need to be investigated. In particular, it is unknown whether the cache complexity of cache-aware algorithms is inherently lower than the complexity of cache-oblivious algorithms. It would be nice to find a separation between the two classes, as well as a simulation result that shows how to make any cache-aware algorithm cache-oblivious with minimal increase in its cache complexity.

Compiler research. The work of this dissertation inspires two lines of research in compilers.

First, because divide and conquer is such an important technique in portable high-performance programs, we should investigate compiler techniques to unroll recursion, in the same way as current compilers unroll loops.

Second, the FFTW system shows the importance of metaprogramming for high performance, whether it be portable or not. For example, the fastest code for a DFT of size 64 on an Alpha processor is one of FFTW's codelets, which consists of about 2400 lines of code. It would have been very hard to write this code by hand. We should investigate the general idea of a ***metacompiler***, which allows a programmer to write both a program and a metaprogram as done in `genfft`. The programmer should be allowed to express algorithms at a high level, and specify how he or she wants the program to be compiled. I do not expect such a system to be generally applicable, but `genfft` shows that even if the metacompiler works for only one problem, it is still worth the effort.

7.2 Summary

In this dissertation we explored techniques to write fast programs whose high-performance is portable in the face of parallelism, memory hierarchy, and diverse processor architectures.

To write high-performance parallel programs, we developed the Cilk-5 language and system. Cilk provides simple yet powerful constructs for expressing parallelism in an application. Cilk programs run on one processor as efficiently as equivalent sequential programs, and they scale up on multiple processors. Cilk's compilation and runtime strategies, which are inspired by the "work-first principle," are effective for writing portable high-performance parallel programs.

Cache-oblivious algorithms provide performance and portability across platforms with different cache sizes. They are oblivious to the parameters of the memory hierarchy, and yet they use multiple levels of caches asymptotically optimally. In this dissertation, we discussed cache-oblivious algorithms for matrix transpose and multiplication, FFT, and sorting that are asymptotically as good as previously known cache-aware algorithms, and provably optimal for those problems whose optimal cache complexity is known.

The location consistency memory model and the BACKER coherence algorithm are one way to achieve portability in high-performance parallel systems with a memory hierarchy. In this dissertation, we proved good asymptotic performance bounds for Cilk programs that uses location consistency.

Finally, the FFTW library adapts itself to the hardware, and it deals automatically with some of the intricacies of processor architectures. While FFTW does not require machine-specific performance tuning, its performance is comparable with or better than codes that were tuned for specific machines.

Bibliography

[1] S. ADVE AND K. GHARACHORLOO, *Shared memory consistency models: A tutorial*, Tech. Rep. 9512, Rice University, Sept. 1995. `http://www-ece.rice.edu/ece/faculty/Adve/publications/models_tutorial.ps`.

[2] S. V. ADVE AND K. GHARACHORLOO, *Shared memory consistency models: A tutorial*, IEEE Computer, (1996), pp. 66–76.

[3] S. V. ADVE AND M. D. HILL, *Weak ordering - new definition*, in Proceedings of the 17th Annual International Symposium on Computer Architecture, Seattle, Washington, May 1990, pp. 2–14.

[4] A. AGGARWAL, B. ALPERN, A. K. CHANDRA, AND M. SNIR, *A model for hierarchical memory*, in Proceedings of the 19th Annual ACM Symposium on Theory of Computing, May 1987, pp. 305–314.

[5] A. AGGARWAL, A. K. CHANDRA, AND M. SNIR, *Hierarchical memory with block transfer*, in 28th Annual Symposium on Foundations of Computer Science, Los Angeles, California, 12–14 Oct. 1987, IEEE, pp. 204–216.

[6] A. AGGARWAL AND J. S. VITTER, *The input/output complexity of sorting and related problems*, Communications of the ACM, 31 (1988), pp. 1116–1127.

[7] M. AHAMAD, P. W. HUTTO, AND R. JOHN, *Implementing and programming causal distributed shared memory*, in Proceedings of the 11th International Conference on Distributed Computing systems, Arlington, Texas, May 1991, pp. 274–281.

[8] A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company, 1974.

[9] A. V. AHO, R. SETHI, AND J. D. ULLMAN, *Compilers, principles, techniques, and tools*, Addison-Wesley, Mar. 1986.

[10] S. G. AKL AND N. SANTORO, *Optimal parallel merging and sorting without memory conflicts*, IEEE Transactions on Computers, C-36 (1987).

[11] B. ALPERN, L. CARTER, AND E. FEIG, *Uniform memory hierarchies*, in Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science, Oct. 1990, pp. 600–608.

[12] M. AN, J. W. COOLEY, AND R. TOLIMIERI, *Factorization method for crystallographic Fourier transforms*, Advances in Applied Mathematics, 11 (1990), pp. 358–371.

[13] A. W. APPEL AND Z. SHAO, *Empirical and analytic study of stack versus heap cost for languages with closures*, Journal of Functional Programming, 6 (1996), pp. 47–74.

[14] N. S. ARORA, R. D. BLUMOFE, AND C. G. PLAXTON, *Thread scheduling for multiprogrammed multiprocessors*, in Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), Puerto Vallarta, Mexico, June 1998.

[15] ARVIND, *Personal communication*, Jan. 1998.

[16] ARVIND, J. W. MAESSEN, R. S. NIKHIL, AND J. STOY, *Lambda-S: an implicitly parallel lambda-calculus with letrec, synchronization and side-effects*, tech. rep., MIT Laboratory for Computer Science, Nov 1996. Computation Structures Group Memo 393, also available at `http://www.csg.lcs.mit.edu:8001/pubs/csgmemo.html`.

[17] D. H. BAILEY, *FFTs in external or hierarchical memory*, Journal of Supercomputing, 4 (1990), pp. 23–35.

[18] L. A. BELADY, *A study of replacement algorithms for virtual storage computers*, IBM Systems Journal, 5 (1966), pp. 78–101.

[19] M. BELTRAMETTI, K. BOBEY, AND J. R. ZORBAS, *The control mechanism for the Myrias parallel computer system*, Computer Architecture News, 16 (1988), pp. 21–30.

[20] B. N. BERSHAD, M. J. ZEKAUSKAS, AND W. A. SAWDON, *The Midway distributed shared memory system*, in Digest of Papers from the Thirty-Eighth IEEE Computer Society International Conference (Spring COMPCON), San Francisco, California, Feb. 1993, pp. 528–537.

[21] J. BILMES, K. ASANOVIĆ, J. DEMMEL, D. LAM, AND C. CHIN, *PHiPAC: A portable, high-performance, ANSI C coding methodology and its application to matrix multiply*, LAPACK working note 111, University of Tennessee, 1996.

[22] J. BILMES, K. ASANOVIĆ, C. WHYE CHIN, AND J. DEMMEL, *Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology*, in Proceedings of International Conference on Supercomputing, Vienna, Austria, July 1997.

[23] G. E. BLELLOCH, *Programming parallel algorithms*, Communications of the ACM, 39 (1996), pp. 85–97.

[24] M. BLUM AND H. WASSERMAN, *Reflections on the pentium bug*, IEEE Transactions on Computers, 45 (1996), pp. 385–393.

[25] R. D. BLUMOFE, *Executing Multithreaded Programs Efficiently*, PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 1995.

[26] R. D. BLUMOFE, M. FRIGO, C. F. JOERG, C. E. LEISERSON, AND K. H. RANDALL, *An analysis of dag-consistent distributed shared-memory algorithms*, in Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), Padua, Italy, June 1996, pp. 297–308.

[27] R. D. BLUMOFE, M. FRIGO, C. F. JOERG, C. E. LEISERSON, AND K. H. RANDALL, *Dag-consistent distributed shared memory*, in Proceedings of the 10th International Parallel Processing Symposium, Honolulu, Hawaii, Apr. 1996.

[28] R. D. BLUMOFE, C. F. JOERG, B. C. KUSZMAUL, C. E. LEISERSON, K. H. RANDALL, AND Y. ZHOU, *Cilk: An efficient multithreaded runtime system*, in Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), Santa Barbara, California, July 1995, pp. 207–216.

[29] ——, *Cilk: An efficient multithreaded runtime system*, Journal of Parallel and Distributed Computing, 37 (1996), pp. 55–69.

[30] R. D. BLUMOFE AND C. E. LEISERSON, *Scheduling multithreaded computations by work stealing*, in Proceedings of the 35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, Nov. 1994, pp. 356–368.

[31] R. N. BRACEWELL, *The Hartley Transform*, Oxford Press, 1986.

[32] R. P. BRENT, *The parallel evaluation of general arithmetic expressions*, Journal of the ACM, 21 (1974), pp. 201–206.

[33] J. B. CARTER, J. K. BENNETT, AND W. ZWAENEPOEL, *Implementation and performance of Munin*, in Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles, Pacific Grove, California, Oct. 1991, pp. 152–164.

[34] B. L. CHAMBERLAIN, S.-E. CHOI, E. C. LEWIS, C. LIN, L. SNYDER, AND W. D. WEATHERSBY, *The case for high level parallel programmin in zpl*, IEEE Computational Science and Engineering, 5 (1998), pp. 76–86.

[35] S. CHATTERJEE, V. V. JAIN, A. R. LEBECK, AND S. MUNDHRA, *Nonlinear array layouts for hierarchical memory systems*, in Proceedings of the ACM International Conference on Supercomputing, Rhodes, Greece, June 1999.

[36] S. CHATTERJEE, A. R. LEBECK, P. K. PATNALA, AND M. THOTTETHODI, *Recursive array layouts and fast parallel matrix multiplication*, in Proceedings of the Eleventh ACM SIGPLAN Symposium on Parallel Algorithms and Architectures, June 1999.

[37] G.-I. CHENG, M. FENG, C. E. LEISERSON, K. H. RANDALL, AND A. F. STARK, *Detecting data races in Cilk programs that use locks*, in Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), Puerto Vallarta, Mexico, June 1998.

[38] *Cilk-5.2 Reference Manual*, 1998. Available on the Internet from `http://theory.lcs.mit.edu/~cilk`.

[39] D. COMMITTEE, ed., *Programs for Digital Signal Processing*, IEEE Press, 1979.

[40] J. W. COOLEY, P. A. W. LEWIS, AND P. D. WELCH, *The Fast Fourier Transform algorithm and its applications*, IBM Research, (1967).

[41] J. W. COOLEY AND J. W. TUKEY, *An algorithm for the machine computation of the complex Fourier series*, Mathematics of Computation, 19 (1965), pp. 297–301.

[42] T. H. CORMEN, C. E. LEISERSON, AND R. L. RIVEST, *Introduction to Algorithms*, The MIT Press, Cambridge, Massachusetts, 1990.

[43] R. E. CRANDALL AND B. FAGIN, *Discrete weighted transforms and large-integer arithmetic*, Math. Comp., (1994), pp. 305–324.

[44] R. E. CROCHIERE AND A. V. OPPENHEIM, *Analysis of linear digital networks*, Proceedings of the IEEE, 63 (1975), pp. 581–595.

[45] D. E. CULLER, A. SAH, K. E. SCHAUSER, T. VON EICKEN, AND J. WAWRZYNEK, *Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine*, in Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, Santa Clara, California, Apr. 1991, pp. 164–175.

[46] E. W. DIJKSTRA, *Solution of a problem in concurrent programming control*, Communications of the ACM, 8 (1965), p. 569.

[47] M. DUBOIS, C. SCHEURICH, AND F. A. BRIGGS, *Memory access buffering in multiprocessors*, in Proceedings of the 13th Annual International Symposium on Computer Architecture, June 1986, pp. 434–442.

[48] P. DUHAMEL AND M. VETTERLI, *Fast Fourier transforms: a tutorial review and a state of the art*, Signal Processing, 19 (1990), pp. 259–299.

[49] F. ERGÜN, *Testing multivariate linear functions: Overcoming the generator bottleneck*, in Proceedings of the Twenty-Seventh Annual ACM Symposium on the Theory of Computing, Las Vegas, Nevada, jun 1995, pp. 407–416.

[50] M. FEELEY, *Polling efficiently on stock hardware*, in Proceedings of the 1993 ACM SIGPLAN Conference on Functional Programming and Computer Architecture, Copenhagen, Denmark, June 1993, pp. 179–187.

[51] S. I. FELDMAN, D. M. GAY, M. W. MAIMONE, AND N. L. SCHRYER, *A Fortran to C converter*, Tech. Rep. 149, AT&T Bell Laboratories, 1995.

[52] M. FENG AND C. E. LEISERSON, *Efficient detection of determinacy races in Cilk programs*, Theory Comput. Systems, 32 (1999), pp. 301–326.

[53] J. D. FRENS AND D. S. WISE, *Auto-blocking matrix-multiplication or tracking blas3 performance from source code*, in Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Las Vegas, NV, June 1997, pp. 206–216.

[54] M. FRIGO, *The weakest reasonable memory model*, Master's thesis, Massachusetts Institute of Technology, 1998.

[55] ——, *A fast Fourier transform compiler*, in Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, May 1999.

[56] M. FRIGO, C. E. LEISERSON, H. PROKOP, AND S. RAMACHANDRAN, *Cache-oblivious algorithms*. Submitted for publication.

[57] M. FRIGO AND V. LUCHANGCO, *Computation-centric memory models*, in Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), Puerto Vallarta, Mexico, June 1998.

[58] M. FRIGO, K. H. RANDALL, AND C. E. LEISERSON, *The implementation of the Cilk-5 multithreaded language*, in Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 1998.

[59] M. Galassi, J. Davies, J. Theiler, B. Gough, R. Priedhorsky, G. Jungman, and M. Booth, *GNU Scientific Library—Reference Manual*, 1999.

[60] G. R. Gao and V. Sarkar, *Location consistency: Stepping beyond the barriers of memory coherence and serializability*, Tech. Rep. 78, McGill University, School of Computer Science, Advanced Compilers, Architectures, and Parallel Systems (ACAPS) Laboratory, Dec. 1993. Revised December 31, 1994. Available at `ftp://ftp-acaps.cs.mcgill.ca`.

[61] ——, *Location consistency: Stepping beyond memory coherence barrier*, in Proceedings of the 1995 International Conference on Parallel Processing, Oconomowoc, Wisconsin, August 1995, pp. 73–76.

[62] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine*, The MIT Press, Cambridge, Massachusetts, 1994.

[63] K. Gharachorloo, *Memory Consistency Models for Shared-Memory Multiprocessors*, PhD thesis, Department of Electrical Engineering, Stanford University, Dec. 1995.

[64] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, *Memory consistency and event ordering in scalable shared-memory multiprocessors*, in Proceedings of the 17th Annual International Symposium on Computer Architecture, Seattle, Washington, June 1990, pp. 15–26.

[65] P. B. Gibbons and E. Korach, *On testing cache-coherent shared memories*, in Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures, Cape May, NJ, 1994, pp. 177–188.

[66] P. B. Gibbons and M. Merritt, *Specifying nonblocking shared memories*, in Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures, 1992, pp. 306–315.

[67] P. B. Gibbons, M. Merritt, and K. Gharachorloo, *Proving sequential consistency of high-performance shared memories*, in Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures, 1991, pp. 292–303.

[68] S. C. Goldstein, K. E. Schauser, and D. E. Culler, *Lazy threads: Implementing a fast parallel call*, Journal of Parallel and Distributed Computing, 37 (1996), pp. 5–20.

[69] G. H. Golub and C. F. van Loan, *Matrix Computations*, Johns Hopkins University Press, 1989.

[70] J. R. Goodman, *Cache consistency and sequential consistency*, Tech. Rep. 61, IEEE Scalable Coherent Interface (SCI) Working Group, Mar. 1989.

[71] R. L. Graham, *Bounds on multiprocessing timing anomalies*, SIAM Journal on Applied Mathematics, 17 (1969), pp. 416–429.

[72] D. Grunwald, *Heaps o' stacks: Time and space efficient threads without operating system support*, Tech. Rep. CU-CS-750-94, University of Colorado, Nov. 1994.

[73] D. Grunwald and R. Neves, *Whole-program optimization for time and space efficient threads*, in Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Cambridge, Massachusetts, Oct. 1996, pp. 50–59.

[74] S. K. S. GUPTA, C. HUANG, P. SADAYAPPAN, AND R. W. JOHNSON, *A framework for generating distributed-memory parallel programs for block recursive algorithms*, Journal of Parallel and Distributed Computing, 34 (1996), pp. 137–153.

[75] R. H. HALSTEAD, JR., *Implementation of Multilisp: Lisp on a multiprocessor*, in Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming, Austin, Texas, August 1984, pp. 9–17.

[76] ——, *Multilisp: A language for concurrent symbolic computation*, ACM Transactions on Programming Languages and Systems, 7 (1985), pp. 501–538.

[77] P. H. HARTEL AND W. G. VREE, *Arrays in a lazy functional language—a case study: the fast Fourier transform*, in Arrays, functional languages, and parallel systems (ATABLE), G. Hains and L. M. R. Mullin, eds., June 1992, pp. 52–66.

[78] E. A. HAUCK AND B. A. DENT, *Burroughs' B6500/B7500 stack mechanism*, Proceedings of the AFIPS Spring Joint Computer Conference, (1968), pp. 245–251.

[79] J. L. HENNESSY AND D. A. PATTERSON, *Computer Architecture: a Quantitative Approach*, Morgan Kaufmann, San Francisco, CA, second ed., 1996.

[80] HIGH PERFORMANCE FORTRAN FORUM, *High performance Fortran language specification v. 2.0*, Jan. 1997.

[81] M. D. HILL, *Multiprocessors should support simple memory consistency protocols*, IEEE Computer, 31 (1998).

[82] J.-W. HONG AND H. T. KUNG, *I/O complexity: the red-blue pebbling game*, in Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing, Milwaukee, 1981, pp. 326–333.

[83] IBM AND MOTOROLA, *PowerPC 604e user's manual*.

[84] L. IFTODE, J. P. SINGH, AND K. LI, *Scope consistency: A bridge between release consistency and entry consistency*, in Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), Padua, Italy, June 1996, pp. 277–287.

[85] C. F. JOERG, *The Cilk System for Parallel Multithreaded Computing*, PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Jan. 1996.

[86] H. W. JOHNSON AND C. S. BURRUS, *The design of optimal DFT algorithms using dynamic programming*, IEEE Transactions on Acoustics, Speech and Signal Processing, 31 (1983), pp. 378–387.

[87] K. L. JOHNSON, M. F. KAASHOEK, AND D. A. WALLACH, *CRL: High-performance all-software distributed shared memory*, in Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles, Copper Mountain Resort, Colorado, Dec. 1995, pp. 213–228.

[88] E. G. C. JR. AND P. J. DENNING, *Operating Systems Theory*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1973.

[89] P. KELEHER, A. L. COX, S. DWARKADAS, AND W. ZWAENEPOEL, *TreadMarks: Distributed shared memory on standard workstations and operating systems*, in USENIX Winter 1994 Conference Proceedings, San Francisco, California, Jan. 1994, pp. 115–132.

[90] P. KELEHER, A. L. COX, AND W. ZWAENEPOEL, *Lazy release consistency for software distributed shared memory*, in Proceedings of the 19th Annual International Symposium on Computer Architecture, May 1992.

[91] D. E. KNUTH, *Sorting and Searching*, vol. 3 of The Art of Computer Programming, Addison-Wesley, second ed., 1973.

[92] ——, *Seminumerical Algorithms*, vol. 2 of The Art of Computer Programming, Addison-Wesley, 3rd ed., 1998.

[93] C. H. KOELBEL, D. B. LOVEMAN, R. S. SCHREIBER, J. GUY L. STEELE, AND M. E. ZOSEL, *The High Performance Fortran Handbook*, The MIT Press, 1994.

[94] D. A. KRANZ, R. H. HALSTEAD, JR., AND E. MOHR, *Mul-T: A high-performance parallel Lisp*, in Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation, Portland, Oregon, June 1989, pp. 81–90.

[95] N. A. KUSHMAN, *Performance nonmonotonicities: A case study of the UltraSPARC processor*, Master's thesis, MIT Department of Electrical Engineering and Computer Science, June 1998.

[96] L. LAMPORT, *How to make a multiprocessor computer that correctly executes multiprocess programs*, IEEE Transactions on Computers, C-28 (1979), pp. 690–691.

[97] J. R. LARUS, B. RICHARDS, AND G. VISWANATHAN, *LCM: Memory system support for parallel language implementation*, in Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, Oct. 1994, pp. 208–218.

[98] F. T. LEIGHTON, *Introduction to Parallel Algorithms and Architectures: Arrays · Trees · Hypercubes*, Morgan Kaufmann Publishers, San Mateo, California, 1992.

[99] X. LEROY, *The Objective Caml system release 2.00*, Institut National de Recherche en Informatique at Automatique (INRIA), August 1998.

[100] E. C. LEWIS, C. LIN, AND L. SNYDER, *The implementation and evaluation of fusion and contraction in array languages*, in Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation, jun 1998, pp. 50–59.

[101] P. LISIECKI AND A. MEDINA. Personal communication, 1998.

[102] C. V. LOAN, *Computational Frameworks for the Fast Fourier Transform*, SIAM, Philadelphia, 1992.

[103] C. LU, J. W. COOLEY, AND R. TOLIMIERI, *FFT algorithms for prime transform sizes and their implementations on VAX, IBM3090VF, and IBM RS/6000*, IEEE Transactions on Signal Processing, 41 (1993), pp. 638–647.

[104] V. LUCHANGCO, *Precedence-based memory models*, in Eleventh International Workshop on Distributed Algorithms, no. 1320 in Lecture Notes in Computer Science, Springer-Verlag, 1997, pp. 215–229.

[105] N. LYNCH AND M. TUTTLE, *Hierarchical correctness proofs for distributed algorithms*, in 6th Annual ACM Symposium on Principles of Distributed Computation, August 1987, pp. 137–151.

[106] W. L. LYNCH, B. K. BRAY, AND M. J. FLYNN, *The effect of page allocation on caches*, in MICRO-25 Conference Proceedings, dec 1992, pp. 222–225.

[107] C. MARLOWE, *The Tragical History of Doctor Faustus*, 1604. A-Text.

[108] J. A. MARUHN, *FOURGEN: a fast Fourier transform program generator*, Computer Physics Communications, 12 (1976), pp. 147–162.

[109] P. MEHROTRA AND J. V. ROSENDALE, *The BLAZE language: A parallel language for scientific programming*, Parallel Computing, 5 (1987), pp. 339–361.

[110] J. S. MILLER AND G. J. ROZAS, *Garbage collection is fast, but a stack is faster*, Tech. Rep. Memo 1462, MIT Artificial Intelligence Laboratory, Cambridge, MA, 1994.

[111] R. C. MILLER, *A type-checking preprocessor for Cilk 2, a multithreaded C language*, Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1995.

[112] E. MOHR, D. A. KRANZ, AND R. H. HALSTEAD, JR., *Lazy task creation: A technique for increasing the granularity of parallel programs*, IEEE Transactions on Parallel and Distributed Systems, 2 (1991), pp. 264–280.

[113] J. MOSES, *The function of FUNCTION in LISP or why the FUNARG problem should be called the envronment problem*, Tech. Rep. memo AI-199, MIT Artificial Intelligence Laboratory, June 1970.

[114] R. MOTWANI AND P. RAGHAVAN, *Randomized Algorithms*, Cambridge University Press, 1995.

[115] S. S. MUCHNICK, *Advanced Compiler Design Implementation*, Morgan Kaufmann, 1997.

[116] T. NGO, L. SNYDER, AND B. CHAMBERLAIN, *Portable performance of data parallel languages*, in Proceesings of the SC'97: High Performance Networking and Computing, 1997.

[117] R. NIKHIL, ARVIND, J. HICKS, S. ADITYA, L. AUGUSTSSON, J. MAESSEN, AND Y. ZHOU, *pH language reference manual, version 1.0*, Tech. Rep. CSG-Memo-369, MIT Computation Structures Group, Jan. 1995.

[118] R. S. NIKHIL, *Parallel Symbolic Computing in Cid*, in Proc. Wkshp. on Parallel Symbolic Computing, Beaune, France, Springer-Verlag LNCS 1068, October 1995, pp. 217–242.

[119] R. S. NIKHIL AND ARVIND, *Id: a language with implicit parallelism*, in A Comparative Study of Parallel Programming Languages: The Salishan Problems, J. Feo, ed., Elsevier Science Publishers, 1990.

[120] M. H. NODINE AND J. S. VITTER, *Deterministic distribution sort in shared and distributed memory multiprocessors*, in Proceedings of the Fifth Symposium on Parallel Algorithms and Architectures, Velen, Germany, 1993, pp. 120–129.

[121] A. V. OPPENHEIM AND R. W. SCHAFER, *Discrete-time Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ 07632, 1989.

[122] W. PARTAIN, *The `nofib` benchmark suite of Haskell programs*, in Functional Programming, J. Launchbury and P. M. Sansom, eds., Springer Verlag, 1992, pp. 195–202.

[123] F. PEREZ AND T. TAKAOKA, *A prime factor FFT algorithm implementation using a program generation technique*, IEEE Transactions on Acoustics, Speech and Signal Processing, 35 (1987), pp. 1221–1223.

[124] *Proceedings of the ACM SIGPLAN '99 conference on programming language design and implementation (PLDI)*, May 1999.

[125] H. PROKOP, *Cache-oblivious algorithms*, Master's thesis, Massachusetts Institute of Technology, June 1999.

[126] C. M. RADER, *Discrete Fourier transforms when the number of data samples is prime*, Proc. of the IEEE, 56 (1968), pp. 1107–1108.

[127] K. H. RANDALL, *Cilk: Efficient Multithreaded Computing*, PhD thesis, Massachusetts Institute of Technology, 1998.

[128] S. K. REINHARDT, J. R. LARUS, AND D. A. WOOD, *Tempest and Typhoon: User-level shared memory*, in Proceedings of the 21st Annual International Symposium on Computer Architecture, Chicago, Illinois, Apr. 1994, pp. 325–336.

[129] J. E. SAVAGE, *Extending the Hong-Kung model to memory hierarchies*, in Computing and Combinatorics, D.-Z. Du and M. Li, eds., vol. 959 of Lecture Notes in Computer Science, Springer Verlag, 1995, pp. 270–281.

[130] D. J. SCALES AND M. S. LAM, *The design and evaluation of a shared object system for distributed memory machines*, in Proceedings of the First Symposium on Operating Systems Design and Implementation, Monterey, California, Nov. 1994, pp. 101–114.

[131] I. SELESNICK AND C. S. BURRUS, *Automatic generation of prime length FFT programs*, IEEE Transactions on Signal Processing, (1996), pp. 14–24.

[132] R. C. SINGLETON, *An algorithm for computing the mixed radix fast Fourier transform*, IEEE Transactions on Audio and Electroacoustics, AU-17 (1969), pp. 93–103.

[133] D. D. SLEATOR AND R. E. TARJAN, *Amortized efficiency of list update and paging rules*, Communications of the ACM, 28 (1985), pp. 202–208.

[134] M. SNIR, S. OTTO, S. HUSS-LEDERMAN, D. WALKER, AND J. DONGARRA, *MPI: The Complete Reference*, MIT Press, 1995.

[135] H. V. SORENSEN, M. T. HEIDEMAN, AND C. S. BURRUS, *On computing the split-radix FFT*, IEEE Transactions on Acoustics, Speech and Signal Processing, 34 (1986), pp. 152–156.

[136] H. V. Sorensen, D. L. Jones, M. T. Heideman, and C. S. Burrus, *Real-valued fast Fourier transform algorithms*, IEEE Transactions on Acoustics, Speech, and Signal Processing, ASSP-35 (1987), pp. 849–863.

[137] P. Stenström, *VLSI support for a cactus stack oriented memory organization*, in Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences, volume 1, Jan. 1988, pp. 211–220.

[138] V. Strassen, *Gaussian elimination is not optimal*, Numerische Mathematik, 14 (1969), pp. 354–356.

[139] P. N. Swarztrauber, *Vectorizing the FFTs*, Parallel Computations, (1982), pp. 51–83. G. Rodrigue ed.

[140] C. Temperton, *Implementation of a self-sorting in-place prime factor FFT algorithm*, Journal of Computational Physics, 58 (1985), pp. 283–299.

[141] ——, *A new set of minimum-add small-n rotated DFT modules*, Journal of Computational Physics, 75 (1988), pp. 190–198.

[142] ——, *A generalized prime factor FFT algorithm for any $n = 2^p3^q5^r$*, SIAM Journal on Scientific and Statistical Computing, 13 (1992), pp. 676–686.

[143] S. Toledo, *Locality of reference in LU decomposition with partial pivoting*, SIAM Journal on Matrix Analysis and Applications, 18 (1997), pp. 1065–1081.

[144] R. Tolimieri, M. An, and C. Lu, *Algorithms for Discrete Fourier Transform and Convolution*, Springer Verlag, 1997.

[145] L. G. Valiant, *A bridging model for parallel computation*, Communications of the ACM, 33 (1990), pp. 103–111.

[146] T. Veldhuizen, *Using C++ template metaprograms*, C++ Report, 7 (1995), pp. 36–43. Reprinted in C++ Gems, ed. Stanley Lippman.

[147] J. S. Vitter, *External memory algorithms and data structures*, in External Memory Algorithms and Visualization, J. Abello and J. S. Vitter, eds., DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society Press, Providence, RI, 1999.

[148] J. S. Vitter and M. H. Nodine, *Large-scale sorting in uniform memory hierarchies*, Journal of Parallel and Distributed Computing, 17 (1993), pp. 107–114.

[149] J. S. Vitter and E. A. M. Shriver, *Algorithms for parallel memory I: Two-level memories*, Algorithmica, 12 (1994), pp. 110–147.

[150] ——, *Algorithms for parallel memory II: Hierarchical multilevel memories*, Algorithmica, 12 (1994), pp. 148–169.

[151] P. Wadler, *How to declare an imperative*, ACM Computing Surveys, 29 (1997), pp. 240–263.

[152] S. Winograd, *On the algebraic complexity of functions*, Actes du Congrès International des Mathématiciens, 3 (1970), pp. 283–288.

[153] ——, *On computing the discrete Fourier transform*, Mathematics of Computation, 32 (1978), pp. 175–199.

[154] L. WITTGENSTEIN, *Tractatus logico-philosophicus*, Routledge and Kegan Paul Ltd, London, 1922.