Diplomarbeit

Parallelism with Asynchronous Lambdas on Fiasco.OC/L4Re

Jan Bierbaum

13. Mai 2013

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer:	Prof. Dr. rer. nat. Hermann Härtig
Betreuende Mitarbeiter:	Dipl.-Inf. Marcus Hähnel
	Dr.-Ing. Marcus Völp

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbständig erstellt und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Dresden, den 13. Mai 2013

Jan Bierbaum

Task

Grand Central Dispatch (GCD) provides modern applications an easy to use abstraction for task level parallelism: blocks and dispatch queues. Its implementation is based on threads and scheduler activations offered by the operating system.
The goal of this diploma thesis is to port a GCD implementation to the Fiasco.OC microkernel and L4Re operating system and show how applications may benefit from blocks and dispatch queues.
For the latter, a complex application with intrinsic dynamic parallelism such as a database or a network stack should be adjusted to make use of the GCD provided mechanisms and evaluated against the base implementation that does not expose this knowledge about fine grain parallelism to the underlying runtime and operating system.
From the results of this analysis, the thesis shall explore the potential for close interaction between applications and scheduling, such as integrating blocks and queues as first class scheduling primitives in the kernel.

Contents

List of Figures

List of Tables

List of Listings

1. Introduction

Multithreading, that is the use of multiple independent flows of execution within a single task, has a long-standing tradition in computing. Although standard systems have supported multi-threading for decades, it was of minor importance for average application developers because commodity hardware usually had just a single CPU. Back then threads were used to improve the responsiveness of the system rather than its performance. The main exception was the field of *high-performance computing* (HPC) with its powerful multi-processor systems where experienced experts strived to develop programs that deliver utmost performance. These programs were optimized for and run on specific, well-known machines. Nowadays this situation has changed.

Multi-Core CPUs In accordance with Moore's Law, the complexity of integrated circuits has been doubling approximately every 18 months since the 1950s [Moo65; Mol06]. For almost four decades this increase (or rather the corresponding decrease in feature size) directly translated into a higher clock rate of the CPU. Thus all programs benefited immediately from new hardware without any need for adoption. But higher clock rates come with higher thermal losses, so this development was bound to grind to a halt sooner or later—a phenomenon also known as "hitting the power wall". And indeed, in recent years we saw an increase in the number of processing cores instead of ever higher clock rates [Sut05].

With the advent of multi-core CPUs in consumer hardware down to small-sized, mobile computers (better known as *smartphones* and *tablets*), multi-threading has finally entered the world of everyday computing devices. Of course, multiple tasks running on a single machine will still benefit from additional CPU cores as each test may be run on a dedicated core. Developers, who want a single application to exploit the full power of this modern hardware, however, are now forced to enter into the indeterministic world of task-level parallelism [Sut05].

Traditional Multi-Threading While there are multiple, readily available technologies to introduce task-level parallelism (see section 2.2), handling concurrency correctly is not an easy task at all. Even established experts sometimes fail to meet its challenges [Lee06]:

> "The portion of the kernel that ensured a consistent view of the program structure was written in early 2000, design reviewed to yellow, and code reviewed to green. The reviewers included concurrency experts, not just inexperienced graduate students (Christopher Hylands (now Brooks), Bart Kienhuis, John Reekie, and myself were all reviewers). We wrote regression tests that achieved 100 percent code coverage. The nightly build and regression tests ran on a two processor SMP machine, which exhibited different

> thread behavior than the development machines, which all had a single processor. The Ptolemy II system itself began to be widely used, and every use of the system exercised this code. No problems were observed until the code deadlocked on April 26, 2004, four years later."

However, this is not only about correctness. As stated by Amdahl's Law [Amd67] the non-parallel parts of an application limit its potential performance gain from additional CPU cores. Thus inherent concurrency should be exposed to the system by explicitly executing code in multiple threads. Unfortunately, there are two major drawbacks: First, the developer is required to split up the workload, divide it among available threads and take care of the necessary synchronization. Second, he needs to ensure the "right" number of threads is used at all times. Alas, that optimum depends heavily on both the hardware as well as the current load of the system. While an insufficient number of threads will leave CPUs idle, too many will impose extra scheduling and synchronization overhead.

How is an application developer supposed to handle this complexity? Lee advocates a radically changed approach toward concurrency to address this problem [Lee06]:

> "Threads must be relegated to the engine room of computing, to be suffered only by expert technology providers."

Operating System Perspective Not only application developers who have to cope with dramatic hardware changes, but operating systems are facing new challenges as well: On the one hand the operating system is expected to provide applications with all the processing power the underlying hardware has to offer, on the other hand, energy consumption gets more and more important.

Especially mobile devices are pushed hard to use their limited battery power as economically as possible. Because of that, these systems have started to incorporate various kinds of specialized, highly efficient processors besides the CPU: A GPU for graphics and video processing is a standard component in today's mobile devices as is a DSP for mobile network communication and telephony. This conglomeration of specialized processors is referred to as *heterogeneous computing* and can be taken to extreme levels by utilizing FPGAs, which allow for dynamic reconfiguration at runtime according to the software's current requirements [Bro+10; Bur+97]. In order to make best use of this variety of processors the system's scheduler needs as much information as possible about the inherent concurrency of the tasks it has to manage.

Grand Central Dispatch *Grand Central Dispatch* (GCD) is a concurrency library that tries to shift the management of threads from the application developer to the operating system, thus complying with Lee's previously cited postulation. The operating system inherently possesses much more information about the hardware and the current load situation than any single user application possibly could. In addition, operating system developers are likely to have much more experience with concurrent programming than the average application developer. With GCD developers no longer orchestrate threads manually, instead they just split their application into small pieces of work and hand

these over to the library for execution. All the tedious chores of thread management and load balancing are taken care of by the library. Close collaboration with the scheduler of the system allows for dynamic adaptation of used threads during runtime according to overall system load.

As the effort to express concurrency in terms of GCD's queue concept is significantly smaller then handling threads directly, developers are more likely to expose inherent parallelism in their applications to the operating system. Block literals, a related technology that introduces anonymous functions into the family of C languages, make the required split up of the application even more convenient and thus encourage the developer to create smaller blocks (quiet literally) of concurrently executing code, which would otherwise have been run in sequence. Although the resulting performance improvement for a single piece of code that has been parallelized this way may be small, according to Amdahl's Law, the overall effect of reducing the serial portion of an application can have significant impact.

1.1. Objectives and Structure of this Work

The main objective of my thesis is to port GCD and the closely related Blocks to the *L4 runtime environment* (L4Re) with the underlying Fiasco.OC microkernel. Once these technologies are available on L4Re, I want to give a first impression of GCD's suitability for practical use as well as an assessment of its overall performance.

Although Fiasco.OC provides native threads and L4Re comes with an implementation of the standard *POSIX Threads* (PThreads) library, development of multi-threaded applications with these low-level instruments is inherently difficult and error-prone. I expect that GCD with its intuitive interface centered around Dispatch Queues will not only lower the entry barriers for newcomers to the field of concurrency but also encourage overall adoption of parallel programming: Dispatching a Block to one of GCD's Concurrent Queues is undoubtedly much more convenient than to create a set of threads and the subsequently arrange for the distribution of work and data. The same line of argumentation is valid for synchronization issues. Chapter 2 introduces all these concepts in proper detail and also provides a brief overview of common techniques—both traditional and novel—to establish task-level concurrency.

GCD and its supporting libraries are available as GNU/Linux versions, which I used as the starting point for my own work. In chapter 3 I will discuss the inner workings of these libraries in greater detail before I subsequently present my L4Re implementations, the changes I had to make during the porting process and my reasons for doing so.

Due to the additional layers of abstraction the comforts that GCD provides will probably have to be paid for with performance losses in comparison to the use of traditional threads. On the other hand, GCD may well lead to a performance improvement when previously sequential parts of an application are parallelized. In an effort to investigate these contradicting assumptions and, furthermore, assess the magnitude of the effects, I conducted a series of measurements. For these, I took two existing applications built on PThreads and remodeled them to use GCD instead. Chapter 4

describes this transformation process: Starting with the troubles of finding appropriate applications, to the problems I encountered during the remodeling and lessons learned over its course.

After the remodeling was complete, I benchmarked both versions of the applications, so as to use the differences in their respective runtimes as a first indication of the implications GCD has on performance. These measurements are the subject of chapter 5, which additionally discusses other important non-functional effects of GCD, such as the library's influence on the code base of an application that is built on top of it.

Finally, chapter 6 concludes this thesis with a summary and a few ideas about possible enhancements of the current GCD infrastructure on L4Re and other follow-up projects that could be arising from this work.

2. Background

This chapter discusses related work and introduces all the background information that is necessary to follow this thesis. First, I will very briefly describe the concept of microkernels and the platform this work is based on, which consists of Fiasco.OC and L4Re. After an overview of the common technologies used to express parallelism in applications, I present GCD and the closely related Block literals as an alternative approach to the direct use of threads to express concurrency. The chapter closes with a short exploration of the technologies GCD is built on—namely KQueue and PThread-Workqueue.

2.1. Fiasco.OC and L4Re

The traditional approach in operating system design is to have a so-called *monolithic kernel* that provides a multitude of services to user-space applications. To do so, the kernel must include non-essential code such as device drivers, file systems, network stacks, and so on. In order to protect the kernel from malicious requests, user applications can invoke the kernel only through specified, well-defined access points known as *system calls*. System calls are an interface about as low-level as can be, however, which makes them cumbersome to use directly. That is why system calls are usually wrapped within library functions that provide much easier access.

All common operating systems (Windows, GNU/Linux, Mac OS X, BSD ...) use this scheme, but it has its downsides as well. The various components of a monolithic kernel share a single protection domain; therefore an error in one component may easily spread to other parts of the kernel and can thus compromise the whole system. Especially device drivers, which constitute a major part of code in monolithic kernels, are known to be error prone [Cho+01].

2.1.1. Microkernels

In contrast to monolithic kernels microkernels follow the paradigm of minimalism in order to reduce the complexity of their code base and thus the possibility of errors. An ideal microkernel has no built-in policies and comprises only functionality that is impossible to implement in user space or when doing so would severely degrade the system's performance. Therefore the L4 family of microkernels limits itself to three basic mechanisms [Lie96]:

Task/Address Space Isolation and resource management
Thread Independent flow of control within an address space
IPC Communication and synchronization between threads

2.1.2. Fiasco.OC

Fiasco.OC [Dreb; LW09], a microkernel developed at TU Dresden, descends from the L4 line but introduces a capability system to implement a local naming scheme. This change allows for improved control over the communication among system components and also reduces the kernel interface drastically to only a single system call—the invocation of a capability. Fiasco.OC provides full support for multi-processor systems and hardware-assisted virtualization.

2.1.3. L4Re

As already pointed out, direct interaction with a kernel is very tedious and this is even truer for a microkernel with its very limited set of features. Because of that, Fiasco.OC comes with the *L4 runtime environment* (L4Re) [Drea], a base system that provides services on a much higher level of abstraction than the microkernel itself, such as loading and running executables. Nevertheless, the system avoids central servers in favor of task-local implementations of services. L4Re also contains standard programming libraries such as uClibc, the C++ STL and most notably PThreads.

2.2. Exploiting Parallelism in Tasks

Today, the most common way to introduce task-level parallelism into an application is through the use of threads. Threads are provided by the kernel of the underlying operating system and represent independent flows of control inside an address space. The memory shared by all the threads of a single task serves as a medium for communication. Whereas this shared memory is immediately available for direct use by all threads, accesses to this common resource must be properly synchronized in order to prevent data inconsistencies. On this account, all the technologies I present in this section provide synchronization mechanisms. Still, it is up to the application developer to actually use those mechanisms and to use them wisely: Insufficient synchronization, on the one hand, will lead to bugs, namely race conditions, that are very hard to locate due to their indeterministic nature. Excessive synchronization, on the other hand, imposes unnecessary overhead and may severely degrade performance.

In the next sections I will briefly present some common mechanism for introducing threads into an application. However, the following topics are *not* addressed:

Non-imperative programming languages Many declarative programming languages—for example functional languages such as Haskell and Erlang—do not allow functions to have side effects or depend on state outside of the function itself. Instead, these languages' functions are similar to mathematical functions and represent a fixed mapping from a set of input parameters to an output. This property (known as *referential transparency*) guarantees that all data dependencies are explicit in the parameters and in the return value of the application's functions. There is no need to synchronize access to shared data simply because there is no shared data, which simplifies (automatic) parallelization tremendously.

Non-kernel threads Threads that are implemented on top of a single kernel or hardware thread do not add any concurrency to the application. Implementations of this type have to rely on the cooperation of all threads involved. Without access to the kernel mode of the CPU, they must resort to non-preemptive scheduling mechanisms. Commonly used names are "user mode threads", "light-weight processes" or "fibers".

Parallelism above task level A typical approach to introduce concurrency into an application is splitting it into multiple tasks. For example, the use of a master–slave model allows a central master task to distribute work to a (possibly dynamic) number of slave tasks. Each slave task will carry out the work assigned to it and provide the master with the results.

Another example is pipelining, which is very typical of UNIX systems, where the output of one task serves as input for another. This continues for a number of stages that together implement some complex processing scheme. The popular MPI protocol also falls into this category as it addresses parallelism on a much larger scale than a single task or even a single machine.

Parallelism at instruction level State-of-the-art CPUs usually provide a set of SIMD operations that process multiple pieces of data at the same time, such as the various *streaming SIMD extensions* (SSE) of the x86 architecture. A compiler that supports automatic vectorization, for example, can perform loop unrolling in combination with the SIMD operations available in modern processors [LCD91].

Although each of the aforementioned approaches has pros and cons of its own, a detailed discussion is outside the scope of this thesis. Instead, I will focus on the type of task-level concurrency that is implemented with C or C++, the prevalent languages in Fiasco.OC and L4Re.

2.2.1. Modularization

The by far most intuitive approach to manage threads in an application is modularization. Multiple threads within a single task can be used the same way as multiple tasks, for example to implement a master–slave model. The major difference is the address space shared among all threads—with all its merits and problems. Strict modularization limits the interaction between the threads of an application and less interaction means less need for synchronization. This, in turn, limits the program's complexity and simplifies both development and maintenance.

The downside of modularization is its coarseness: Because this design restricts interaction among threads, additional threads will usually only be created to handle larger pieces of work, not supposedly small processing tasks. However, these smaller tasks can add up to a significant workload and may very well run in parallel, assuming that they are independent of each other.

2.2.2. Libraries

There are multiple libraries that provide access to threads on very different levels of abstraction. I will briefly discuss the more common ones available for use with C or C++.

2.2.2.1. POSIX Threads

The PThreads library stands out because it is part of the POSIX standard (POSIX Threads Extension[1]) [Gro08] and therefore in widespread use. In POSIX-compliant systems it is typically the most basic threading library and often provides the only means to handle threads in applications written in C or C++, prior to C++11 (see section 2.2.3) that is. It is noteworthy that L4Re, too, supports PThreads.

The PThreads library is very low-level and operates on single threads, which are created (`pthread_create`) or terminated (`pthread_exit`, `pthread_cancel`) by the available management functions. The developer can also manipulate the scheduling policies and priorities (`pthread_setschedparam`) applied to each thread.

Synchronization A thread can wait for the termination of another thread (`pthread_join`) or rendezvous with a specified number of threads at a barrier (`pthread_barrier_t`). The latter means that upon reaching the barrier the threads wait until the specified number of threads has arrived at this point. Only then all involved threads will resume execution.

Of course, there are also classical locks, spin locks (`pthread_spinlock_t`) or mutexes (`pthread_mutex_t`), as well as the more potent reader–writer locks (`pthread_rwlock_t`). Although semaphores (`sem_t`) are not strictly a part of PThreads, they are available with the *POSIX Realtime Extension*. Yet another option are *condition variables* (`pthread_cond_t`) that allow a thread to wait for an event explicitly signaled by another thread through said variable. Therefore condition variables are a means to express dependencies between threads or rather between the actions they perform.

To help developers in handling shared memory, PThreads offer *thread-local storage* (`pthread_key_t`). Each thread gets its own private version of such data; actions of one thread on "its" instance of the data does not affect other threads' versions of the data in any way. Last but not least, there is a mechanism for initialization purposes (`pthread_once`) that allows to execute a given function once (and only once!) during the runtime of an application.

2.2.2.2. Intel Threading Building Blocks

Intel Threading Building Blocks (TBB) [Cor] is a high-level C++ library for task parallelism that was first published by Intel in 2006. It is available for all major operating systems and written entirely in standard C++, making heavy use of template code.

[1] Some of the mechanism described in this section are only available if the PThreads implementation in question supports the *Realtime Threads* and *Advanced Realtime Threads* options, which is usually the case nowadays.

TBB is based on the concept of non-preemptive user-level threads, called *Tasks*[2] that run on top of the native threads provided by the operating system. *Task graphs* are used to specify dependencies between the individual Tasks. New Tasks are created from a function, a functor object or a C++11 lambda expression. Alternatively, a user-defined class may directly extend the `task` class and override its abstract `execute` method.

The *Task scheduler* runs Tasks using threads of the operating system and in the process balances the load across all available CPU cores. Various parameters of a Task, for example its priority or the affinity to a particular worker thread, can be customized. Although developers may directly spawn and manage Tasks and Task graphs in this manner, TBB provides a more convenient alternative method in form of assorted concurrent algorithms: function templates (`parallel_for`, `parallel_sort`, `parallel_reduce`, ...) that handle all necessary routine work and conceal the full complexity of the Task scheduler.

Native threads of the underlying operating system are accessible via a platform-independent interface (`thread`) and can safely be mixed with Tasks. The documentation recommends using them whenever operations are expected to block often; input–output operations for example. As Tasks, by design, cannot be preempted, any blocking operation would stall the thread the affected Task runs on.

Synchronization Task graphs are the basic means of synchronization in TBB but the library also comes with traditional locking mechanisms—simple lock (`mutex`), spinlock (`spin_mutex`), reader–writer lock (`*_rw_mutex`). For the sake of compatibility with older compilers, it even reimplements a limited subset of the condition variables and scoped locking of C++11 (see also section 2.2.3 on page 17). In addition, there are enhanced versions of the simple locks, such as a recursive lock (`recursive_mutex`) that may be locked multiple times by a single thread or a lock that guarantees fairness (`queuing_mutex`). However, explicit locking is discouraged, as explained in the previous paragraph. Therefore, a special template class (`atomic`) provides an easy way to upgrade any integral, enumeration or pointer type with atomic operations. These enhanced data types can be used to implement lock-free synchronization schemes.

Task groups represent a number of Tasks that run in parallel. The application may dynamically add new Tasks to a group or wait for the completion of all Tasks associated with a given group. Additionally TBB offers thread-safe container classes (`concurrent_hash_map`, `concurrent_vector`, `concurrent_queue`) and classes for thread-local storage.

2.2.2.3. Open MP

Open Multi-Processing (Open MP) [ARB11], a concurrency API first published in 1997, does not quite fit the pattern of libraries I present in this section because it is not a standalone library but requires compiler support. However, all major compilers (except Clang) come with Open MP support and the nature of the compiler interaction Open MP uses provides for graceful degradation: Any compiler lacking Open MP support will

[2] Please note the difference between a *task*—one of the basic building blocks of an operating system that corresponds to a program in execution—and a *Task*, a user-level thread used by the TBB library. In this thesis, I will use the latter term exclusively in the context of TBB and only in the current section.

just ignore the additional language constructs and produce a fully functional, but non-concurrent program from the source code. Even though Open MP supports C, C++ and FORTRAN, I will not discuss its FORTRAN features for the reasons given at the beginning of this section.

The main instrument of Open MP are `#pragma` directives that are inserted into C or C++ code in order to establish *parallel regions* in otherwise serial code. A useful side effect of this approach is the opportunity to gradually parallelize a serial program by adding parallel regions one after another. Whenever the compiler encounters a parallel region (`#pragma omp parallel`) it employs the so-called *fork–join model*—upon entering the region, Open MP's runtime environment creates a *team*, which is a set of threads with a single master thread. The total number of threads and their mapping to available processors depends on the system load and various configuration variables, which can be set either before the program starts or via Open MP runtime library functions. Once created, a team cannot be adjusted until the parallel region is complete.

Within a parallel region all threads of the team will, by default, simply execute the same code in parallel. Usually that is not what the developer intends, so various *work sharing constructs* change this behavior:

- The iterations of any `for` loop can be distributed among the threads of the team so the loop is executed concurrently (`#pragma omp for`).
- There may be a number of *sections* (`#pragma omp sections`, `#pragma omp section`), each of which is executed by a single member of the team. With the help of this construct multiple mutually independent parts of the program may run at the same time even though each of these parts in itself is sequential.
- Distinct pieces of code (`#pragma omp task`) can be handed over to the Open MP runtime environment for concurrent execution. The effect of this construct is quite similar to that of a thread pool.

The developer can use so-called *clauses*, options attached to a directive, to adjust aspects of the directive's default behavior or to provide the Open MP runtime environment with additional meta information. Clauses can, for example, fine-tune the manner in which the iterations of a parallelized `for` loop are distributed among the thread of a team (`schedule`). They can as well set the number of threads in the team that executes a parallel region (`num_threads`) or completely prevent concurrent execution of a parallel region according to runtime constraints (`if`).

Synchronization Within a parallel region there can be serial sections that are executed by just a single thread of the team—either any available thread (`#pragma omp single`) or the master thread (`#pragma omp master`) in particular.

When it comes to locks, Open MP offers two fundamental implementations. One is via the runtime environment and provides two kinds of locks—the traditional mutex (`omp_lock_t`) and a recursive version that can be locked multiple times by a single thread (`omp_nest_lock_t`)—as well as functions for locking and unlocking. The other option, a directive (`#pragma omp critical`), fits better into Open MP's overall structure:

It establishes named regions of code and Open MP guarantees mutual exclusion of all regions with the same name. In addition, there is a construct (`#pragma omp atomic`) for very short critical sections (restricted to one or two specific statements) that require atomic execution.

The variables used within a parallel region can either be shared by all the threads of the team (`public`) or each thread can have its own version of the variable (`private`). Once again clauses are used to specify this property. Private variables, by definition, require no synchronization. However, the developer is responsible for proper synchronization of any access to global data.

Last but not least, Open MP provides barriers (`#pragma omp barrier`) at which the whole team will rendezvous before resuming work. Any thread that arrives at the barrier before the others will be forced to pause. For this reason barriers can severely compromise performance, especially in the presence of highly dynamic workloads.

2.2.3. Language Support in C++11

With its recent update in 2011, the C++ language standard (called C++11) [Sta11] introduced mechanisms to handle multiple threads. For the first time, the C++ programming language itself defines a memory model and the means to manage both threads and their synchronization. Although most C++ compilers already support major parts of the new standard, to my knowledge there is not a single compiler that implements C++11 completely at the time of writing. However, most of the multi-threading features I discuss here, are already widely available.

`std::thread` is the central entity of the C++11 threading concept. It is implemented within the STL and it represents and controls a single thread of execution. The class's constructor creates an additional thread from a functor, a function pointer or a lambda expression (see also section 2.3.3 on page 20). Optionally, the constructor accepts an arbitrary number of further parameters, which are subsequently passed on to the functor/function/lambda when the new thread starts execution. For non-portable use with platform-specific functions `std::thread` also provides access to the underlying native thread (`std::thread::native_handle`).

Synchronization With the keyword `thread_local` and the associated new storage duration *thread-local* the latest standard of C++ introduces thread-local variables into the core language. All the other synchronization features I discuss in this section are provided by the STL, not by the language itself.

The most trivial kind of synchronization is available directly via `std::thread::join` which allows any thread to wait for the completion of the one that is associated with the respective thread object.

Of course, the STL also comes with all the traditional synchronization primitives. There are four different versions of mutexes: basic (`std::mutex`), recursive (`std::recursive_mutex`), timed (`std::timed_mutex`) and a combination of the latter two (`std::recursive_timed_mutex`). Recursive mutexes can be acquired multiple times by a single thread without deadlocking. Timed mutexes, on the other hand, feature an optional timeout for locking. If the mutex cannot be acquired within this time limit, the locking operation simply fails and does

```
1  // define new Block reference types
2  typedef int  (^int_block)  (int, int);
3  typedef void (^void_block) (void);
4
5  ...
6
7  // define Block literals and store them in Block references
8  int_block  block_1 = ^(int a, int b) { return a + b; };
9  void_block block_2 = ^{ puts("Hello world!"); };
10 ...
11 // copy a Block reference
12 // (without using custom Block reference type 'int_block')
13 int (^block_3) (int, int) = block_1;
14 int (^block_4) (int)      = block_1;  // compiler error
15 ...
16 // invoke Block references
17 printf("Block 1: %d\n", block_1(23,42)); // yields "Block 1: 65"
18 block_2();                               // yields "Hello world!"
19 printf("Block 3: %d\n", block_3(13,7));  // yields "Block 3: 20"
```

Listing 2.1. *Block Literal and Block Reference*

not block the requesting thread indefinitely. Thanks to so-called *locks*—template classes (`std::lock_guard`, `std::unique_lock`) reminiscent of smart pointers—mutexes can be used in accordance with the RAII idiom[3]. A lock's constructor and destructor perform the locking and unlocking respectively, adding both convenience and exception-safety to a bare mutex. Furthermore, function templates (`std::lock`, `std::try_lock`) assist in locking a set of mutexes in an all-or-nothing fashion without the risk of a deadlock.

Whenever one thread needs to wait for an explicit notification from another thread, condition variables (`std::condition_variable*`) are the method of choice. In addition to this very basic form of asynchronous communication between threads, C++11 also offers a one-time communication channel. It consist of a pair of template classes—a write-only "sender" (`std::promise`) and a read-only "receiver" (`std::future`). The receiving thread may also block on the `std::future` until a value is set.

Atomic operations are available in C++11 as well, if only for integral type and pointers. For this purpose the STL provides a class template (`std::atomic`) along with various specializations.

Finally, the STL features a template function (`std::call_once`) that guarantees the one-time execution of a given function even if it is called by multiple threads concurrently. This mechanism facilitates thread-safe initialization of shared data.

2.3. Block Literals

Block literals [Teab], or "Blocks" for short, are an extension of the C language family that introduces the concept of anonymous functions into these programming languages.

[3] RAII stands for "resource acquisition is initialization" and refers to a standard technique for preventing resource leakage in the presence of exceptions.

```
1  int var_1;
2
3  void func(void) {
4    static int var_2;
5    int var_3 = 0;
6    __block int var_4 = 0;
7
8    ^ {
9     int sum = var_1 + var_2 + var_3 + var_4; // ok (sum == 0)
10    var_1 = 42; // ok
11    var_2 = 42; // ok
12    var_3 = 42; // compiler error
13    var_4 = 42; // ok
14   } ();
15 }
```

Listing 2.2. *Variables Captured by a Block Literal*

Even though Blocks are very similar to normal functions there is one crucial difference; each Block preserves its lexical scope when it is defined. Thus Blocks represent what is known as closure or lambda function in other programming languages.

Apple introduced Blocks together with GCD, nevertheless they are a completely independent technology that can also be used on its own. However, as it is a non-standard language extension only two compilers, namely Clang and Apple's fork of GCC, support Blocks at the time of writing.

2.3.1. Block Definition and Block Reference

A Block definition starts with a circumflex, the newly introduced unary Block construction operator. This is followed by the Block's return type and parentheses containing the Block's parameters and finally the body enclosed in curly braces:

```
^ return_type (parameter_types) { definition }
```

Both the return type and the parameter list are optional and default to `void` if omitted. Within the body you may use any code that is legal in the body of an ordinary function, including the declaration of new automatic variables.

Of course, a Block can be invoked immediately upon definition with the usual function call syntax:

```
^ { puts("Hello world!"); }();
```

In most cases that would not make much sense though, as the Block serves no real purpose this way. Instead Blocks are usually stored for later use in a Block reference whose syntax is highly reminiscent of a function pointer:

```
return_type (^name) (parameter_types);
```

Both Block references may also be used in defining a new type with the help of `typedef`. For a full example please have a look at listing 2.1 on the facing page.

2.3.2. Captured Variables

As stated before, Blocks preserve their lexical scope—a mechanism which is referred to as the Block "capturing" all variables accessible at the time of its definition. This means that any such variable is readily available within the body of the Block no matter when the Block is invoked. Variables with static storage duration are captured by reference and may thus be read and written as usual. Automatic variables (commonly referred to as "local variables"), in contrast, are captured by value. They are read-only within the Block's body and any attempt to write a captured variable with automatic storage duration will result in a compile-time error.

The newly introduced storage specifier `__block` offers a third option: A variable with Block storage duration will be preserved as long as any Block references it and may also be written from within a Block's body. Therefore, a Block variable is similar to a static variable in terms of data access from within Blocks but has a more limited lifetime.

Note, that the capturing rules I have just described do not apply to automatic variables declared within a Block itself. Those variables retain their usual properties: They may be read and written and their scope and lifetime are limited to the Block in which they are declared. Listing 2.2 on the previous page shows a comprehensive example of various variables used by a Block literal.

Blocks do *never* synchronize access to shared data. Therefore, within a multi-threaded environment, it is up to the application developer to guarantee proper serialization; see section 2.4.2 for synchronization using GCD.

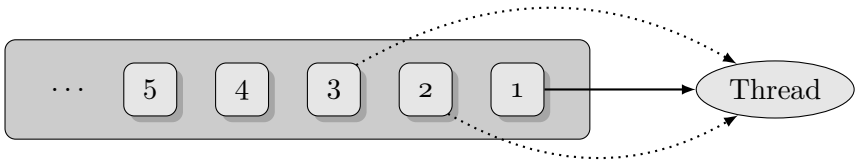
2.3.3. Comparison with C++11 Lambda Expressions

I have already discussed the new concurrency features of C++11 in section 2.2.3, but the standard comes with another interesting functionality: *lambda expressions* or *lambdas* for short [Sta11]. Both, Blocks and C++11 lambdas, define a closure/anonymous function but the use of Blocks is not limited to C++11. They are also compatible with previous versions of C++, with C and with Objective-C. However, as they are a non-standard language extension, actual compiler support for Blocks is currently limited to Clang whereas all major C++ compilers are expected to eventually support C++11.

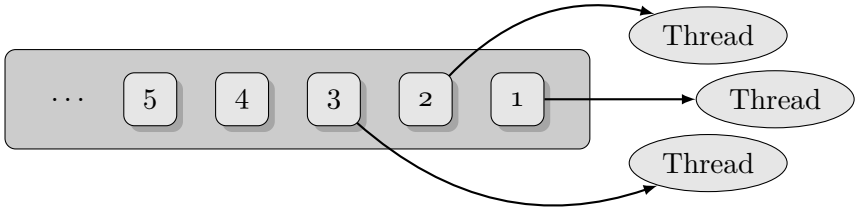
Furthermore, C++11 lambdas are more customizable than Blocks when it comes to captured variables. They accept a detailed specification of variables to capture and which of those to capture by reference or by value. Blocks, on the other hand, use the fixed standard rules explained in section 2.3.2. Although this is a drastic restriction, it simplifies matters for the developer and thus fits well into the general concept of GCD.

2.4. Grand Central Dispatch

Developed by Apple and first introduced with Mac OS X 10.6 in 2009, *Grand Central Dispatch* (GCD) [Teaa] represents an alternative paradigm to express task-level parallelism in applications. The library manages the distribution of work among available CPU cores, the creation and destruction of threads as necessary and, under certain conditions, also synchronization (see section 2.4.2).



(a) *serial*



(b) *concurrent*

Figure 2.1. *Grand Central Dispatch—Dispatch Queues*

Instead of manually handling various threads or assigning jobs to a thread pool, the developer can use GCD to *dispatch* pieces of work—to which for the sake of simplicity and consistency I will from now on uniformly refer as *items*—to so-called Dispatch Queue; the basic interface of GCD. These items are subsequently removed and executed by GCD in a strict FIFO order. Whenever the developer dispatches an item to a queue he has a choice to do so either in a synchronous or asynchronous manner: The asynchronous dispatch (`dispatch_async*`) allows the current item to continue its execution and is thus preferred for achieving a higher degree of concurrency. A synchronous dispatch (`dispatch_sync*`), on the other hand, forces the current item to wait until the newly dispatched item is finished.

There are two[4] methods for specifying such an item: The most common and intuitive one is to use a Block for this purpose. However, as I mentioned in section 2.3, Blocks are not part of any language standard and thus a compiler lacking Block support may quickly thwart this option. An alternative, fully standard-compliant form for queue items is a function pointer (of the type `void (*func)(void*)`) in combination with a matching (`void*`) data parameter. Upon execution of the queue item this parameter is passed on to the function.

2.4.1. Dispatch Queues

Within GCD there are two distinct types of Dispatch Queues, which differ in the way they execute items dispatched to them.

[4] This is true for Apple's official GCD implementation. However, XDispatch [MLB] for example, a fork of the library that incorporates features of C++ into GCD, also accepts C++11 lambda expressions (briefly discussed in section 2.3.3) to specify queue items.

```
1 Mutex mtx_access;
2 ...
3 lock(mtx_access);
4   // access data
5 unlock(mtx_access);
```

(a) PThreads

```
1 dispatch_queue_t q_access =
   ↪ dispatch_queue_create("mutex",
   ↪ DISPATCH_QUEUE_SERIAL);
2 ...
3 dispatch_sync(q_access,
4   ^{ /* access data */ });
```

(b) GCD

Listing 2.3. *PThreads versus GCD—Serialization*

Serial Dispatch Queue Whenever items need to be run mutually exclusive, they should be dispatched to a Serial Queue. In addition to the FIFO property all Dispatch Queues possess, Serial Queues provide the guarantee that only a single queue item is executed at any time. This property makes them a viable option for synchronization purposes as detailed in section 2.4.2.

Concurrent Dispatch Queue The aptly named Concurrent Queue potentially runs all available items in parallel. Items are still taken from the queue in FIFO order, but due to different execution times they may finish in an arbitrary order. GCD provides no guarantee whatsoever about the number of items that are processed at the same time. That rather depends on the number of available CPU cores as well as on the overall system load.

In general, there is no synchronization between the individual items of a Concurrent Queue. The one single exception to this rule are so-called *barriers*, which are discussed along with GCD's other synchronization mechanisms in section 2.4.2.

2.4.2. Synchronization

The concurrency model of GCD differs a lot from traditional threads as it is solely based on Dispatch Queues. Consequently, GCD offers a suitable set of synchronization facilities.

Serial Dispatch Queue Serial Queues, by design, offer synchronized access to any resource, provided that the resource in question is used exclusively via a dedicated Serial Queue. All items in such a queue run one after another, so there is no situation where two items are active at the same time and can race each other.

Apple's guideline on concurrent programming [Inca] strongly discourages the use of locks in the context of GCD: Failure to acquire a lock inevitably blocks the item and thus the underlying worker thread that runs this item. Instead, serial queues replace mutexes and provide the additional benefit that there is no longer an unlock operation that might be forgotten; see Listing 2.3 for a direct comparison.

Synchronous Dispatching If an item is dispatched to any (serial or concurrent) queue in a synchronous manner, this establishes an implicit synchronization between said item and the currently active one, which does the dispatching. The current item's execution

```
1 RW_Lock access;
2 ...
3 lock_r(access);
4     // read data
5 unlock_r(access);
6 ...
7 lock_w(access);
8     // write data
9 unlock_w(access);
```

(a) *PThreads*

```
1 dispatch_queue_t q_access =
   ↪ dispatch_queue_create("rw-lock",
   ↪ DISPATCH_QUEUE_CONCURRENT);
2 ...
3 dispatch_sync(q_access,
4  ^{ /* read data */ });
5 ...
6 dispatch_barrier_sync(q_access,
7  ^{ /* write data */ });
```

(b) *GCD*

Listing 2.4. *PThreads versus GCD—Reader–Writer Synchronization*

will be suspended until the dispatched one is complete. Although this mechanism provides very easy and intuitive means of synchronizing two items, it should be avoided because the worker thread that runs the current item gets stalled. Furthermore, this characteristic leads to a deadlock when an item that is executed in a Serial Queue synchronously dispatches another item to the same queue.

Dispatch Semaphore GCD provides so-called *Dispatch Semaphores* as a light-weight alternative to classical counting semaphores. They have the same semantics but will only invoke the system's kernel when a thread needs to be suspended.

Semaphores should be used sparsely because of the circumstances explained in the first paragraph of this section. Whereas a simple mutex is sufficient for strict mutual exclusion, semaphores are the method of choice whenever at least two (yet not arbitrarily many) items may run concurrently.

Barrier A barrier is a special queue item that is guaranteed to run by itself, even in a Concurrent Queue. Furthermore, GCD guarantees that all items which are already in the queue when a barrier is added will have finished before the barrier starts and any item added after the barrier will not be execute until the barrier item is complete.

Due to this behavior, barrier items can be used to turn a dedicated Concurrent Queue into an ideal replacement for a reader–writer lock; please refer to listing 2.4 for a side-by-side comparison of PThreads and GCD.

Group At times, a given set of queue items needs to be processed completely before some final action can be taken. For example, any computation that relies on multiple, mutually independent pieces of data may prepare these in parallel, yet it cannot proceed until all prerequisites are available: Imagine rendering the pieces of an image concurrently and presenting or saving the image once it is complete. Due to the indeterministic behavior of the items in a Concurrent Queue—items are started in a strict FIFO order but do not necessarily finish in that same order—the final item cannot simply be dispatched by the "last" data processing item. Some other items may run for a long time and thus finish after the "last" one. Although it would be possible to use a Serial Queue for the whole process this would completely prevent any concurrent execution.

For situations like this, GCD provides *Dispatch Groups*. A Group consists of an arbitrary number of queue items. Once all these items have finished execution, the Group automatically schedules another item for execution. Both, this item and its target Dispatch Queue, are specified by the developer. In addition, the application may wait until all items of a Group complete, thereby forcing the waiting item to block its underlying worker thread.

Groups are completely independent from Dispatch Queues. That means a queue item can be a member of any Group regardless of the queue it is dispatched to. Therefore it is perfectly legal to combine items within a Group which are distributed among a multitude of different (serial or concurrent) Dispatch Queues.

2.4.3. Building Blocks

As a high-level library, GCD is built upon two, rather low-level technologies, KQueue and PThread-Workqueue, which I will introduce in the remainder of this chapter.

2.4.3.1. KQueue

In some situations a user application has to react to or wait for certain events that are managed by the kernel. A typical example is an application waiting for input from the user or for the completion of asynchronous file operations[5]. Once such an operation is started, the user application needs to know when it is complete and data may be read or new data may be written.

The naive polling approach imposes considerable overhead as the application keeps querying for the current status. The situation is comparable to busy waiting for a lock variable. While in single-threaded tasks the constant polling monopolizes the only thread available and thus stops the whole application from making any progress, in multi-threaded applications polling does work in principle. Yet, it still requires at least one dedicated thread to perform the querying and the overhead stays prohibitively large (in terms of computing time and, in consequence, energy consumption) for long-lasting asynchronous operations.

To address this issue, the POSIX standard [Gro08] defines the `select` and `poll` standard library functions. Both of these functions allow a user-space application to monitor a specified set of file descriptors for status changes. However, `select` and `poll`, on each invocation, require a full list of file descriptors to monitor and return the complete list marking those descriptors for which an event occurred. It is up to the application to walk the whole list and filter out the marked list entries. Once the number of events an application is interested in gets large, this concept renders both functions inefficient and inconvenient.

KQueues were introduced into the FreeBSD kernel to counter this problem and provide a scalable, yet safe interface for user-space applications to monitor events within the kernel. In fact, KQueues have become much more versatile than `select` or `poll` because

[5] Keep in mind that in monolithic operating systems it is the kernel that implements file system functionality and provides it to user-level applications.

```
1  pthread_workqueue_t wq;
2  void* data;
3  ...
4  void do_work(void* data) { ... }
5  ...
6  // initialize library
7  pthread_workqueue_init_np();
8
9  // create a work queue (default priority, non-overcommit)
10 if (pthread_workqueue_create_np(&wq, NULL)) {
11   // work queue not created -- error handling
12 }
13
14 ...
15
16 // add work item
17 if (pthread_workqueue_additem_np(wq, do_work, data, NULL, NULL)) {
18   // item not added -- error handling
19 }
```

Listing 2.5. *Usage of PThread-Workqueue*

they are not restricted to monitoring file descriptors but also provide, for example, timers which generate events in customizable intervals.

The main difference between `select`/`poll` on the one hand and KQueues on the other is that the latter establish in-kernel data structures that keep track of the events (e.g., file descriptors) to monitor and also maintain information about events that have occurred. With this information statically available within the kernel, the user space application no longer needs to provide a full list of events of interest on each call. This greatly simplifies the use of KQueues and makes them suitable for handling a large number of event sources.

The library's interface is deliberately kept simple: First, a task creates a new KQueue (`kqueue`). A single function (`kevent`) is subsequently used to manage (add/modify/remove/ ...) monitoring requests and, at the same time, wait for and retrieve events. In the special case that there are already events pending when the user application queries a KQueue, `kevent` returns immediately. Otherwise the function blocks the current thread until either an event occurs or the user-specified timeout expires.

Note that a task may create an arbitrary number of KQueues which can be used to monitor distinct (not necessarily disjunct) sets of event sources in different parts of the application at the same time.

2.4.3.2. PThread-Workqueue

PThread-Workqueues are a non-standard extension to the common PThreads library I described in section 2.2.2.1. Their main principle is the concept of work queues, which are filled with work items by the developer and use an automatic, kernel-managed thread pool for processing these items in parallel. In this regard work queues are strikingly similar to GCD's dispatch queues although much less flexible and convenient. Aside

from the missing comfort of Block literals the PThread-Workqueue library offers no equivalent to GCD's serial queues and relies solely on the facilities provided by PThreads for synchronization purposes.

Workqueues, the central infrastructure of PThread-Workqueue, are quite simple. Their interface is reminiscent of `pthread_create`: Once created, a work queue accepts so-called *work items*—sets of two pointers, whereby one pointer refers to a function of type `void (*func)(void*)` to be executed and the other indicates the function's data parameter. See listing 2.5 on the preceding page for a short example of usage of PThread-Workqueues.

The library removes items from a work queue in a strict FIFO order and executes them concurrently by means of an internal thread pool. Depending on the current system load the kernel scheduler may increase or decrease the number of worker threads in this pool. Because PThread-Workqueues are concurrent by design, the developer has to use the instruments of the standard PThreads library to enforce serialization or synchronization if necessary.

Each work queue has two important attributes, both of which are specified when the queue is created and immutable thereafter:

Priority The priority of a work queue is essentially the same as a thread's priority and determines when the queue's work items run: As long as there are items available in a high-priority queue those get scheduled, possibly stalling lower-priority queues in the process.

Overcommit Unlike normal work queues, overcommit queues may spawn new threads at any time—*regardless* of the current system load. Because this behavior can quickly lead to the creation of an extremely large number of worker threads, overcommit queues should be used with care. Note that GCD's Serial Queues (see also section 2.4.1) are implemented on top of overcommit work queues, so the same is true for them.

3. Porting Grand Central Dispatch to L4Re

In this chapter I address the process of porting the GCD infrastructure—that is the library itself and its supporting technologies: Blocks, KQueue and PThread-Workqueue—to L4Re. Each of the following sections (except for the first) covers one of these libraries and discusses selected aspects of their implementation on L4Re. Furthermore, I will explain why and how I deviated from the GNU/Linux implementation of the respective library and point out problems encountered in the course of this work.

3.1. Design Principles

The original implementations of KQueue and PThread-Workqueue—two of the libraries GCD is built on—depend to a large extent on support from the operating system's kernel. As a microkernel, Fiasco.OC strives for minimalism (see also section 2.1) and therefore does not lend itself to the integration of non-essential features such as provided by these libraries. Fortunately both libraries are available under free software licenses and there are ports available that have been adapted to run in the user-space of GNU/Linux. In combination with the PThreads support of L4Re these implementations proofed to be a good starting point for my own work.

Performance versus Updates Early in the porting process I had decided to adopt a "minimally invasive" strategy, that is to modify existing code as little as possible to allow for easy updates whenever the code of the original libraries (or rather their GNU/Linux version) changes. In section 5.1 on page 63 I will elaborate to what extend I managed to comply with this resolution. Whenever the code required adaptation, I took great care to cleanly separate any changes from the original library code; either by wrapping changes with preprocessor directives (`#if`, `#ifdef`) or, in case of larger changes and supplementary code, by creating new source files altogether. This approach makes my modifications easy to make out and facilitates merging them into future library updates.

The most obvious disadvantage of this approach are the extra layers of abstraction that are preserved this way: GCD is built on KQueue and PThread-Workqueue, whose GNU/Linux implementation, in turn, relies on PThreads. Even though this strict layering simplifies porting and updates, it is also very likely to diminish the overall performance of L4Re applications built on top. For this reason, I investigated the option to completely uncouple GCD from its supporting libraries at the very beginning of my work. I will elaborate this line of thought in section 3.3.2 on page 29.

After careful consideration, however, I came to the conclusion that the fast incorporation of future changes and, even more importantly, compatibility with the official release of GCD outweigh raw performance. GCD, in particular, is still evolving fast and my implementation is currently used exclusively in the experimental field.

Optimization Aside from the implementation I also adopted configuration parameters from the GNU/Linux version of each library whenever possible. For the reasons stated in the previous paragraph, I refrained from optimizing these parameters for use with L4Re.

One single exception from this rule is the PThread-Workqueue library: Due to the very basic nature of Fiasco's scheduler I was forced to enhance the GNU/Linux version of the library anyway, because otherwise it would not have been able to utilize more than a single CPU (see also section 3.5.3 on page 43). As PThread-Workqueue plays a central role for the overall performance of GCD, I decided to apply at least some basic optimizations, not all of which resulted in an actual improvement. Please refer to sections 3.5.2 and 3.5.3 on page 38 and on page 43 for more information on this matter. Should GCD prove its usefulness in the L4Re environment, there are options for further optimizations which can be carried out later on—see also section 6.3 on page 82.

Seamless Integration into L4Re L4Re comes with its own build infrastructure. Based on BID [LA06] and comprising an intricate set of scripts this infrastructure allows for the configuration of the system and takes care of dependency resolution among software packages. For easy and familiar usage with other components of L4Re, I have integrated all relevant libraries (Blocks, KQueue and PThread-Workqueue, GCD) into this existing environment. Each library can be linked to applications that use it (in either a static or dynamic manner) and none entails further runtime requirements. The only exception being, once again, PThread-Workqueue and its server mode (see also section 3.5.2.1), which requires a running server.

Building Requirements All but one of the involved libraries (and in particular their GNU/Linux ports) are written in standard C. Therefore they can be built with any C compiler. This time GCD is an exception, because it relies on Blocks for its internal design and thus requires Clang for compilation (see also section 2.3). However, the resulting object files can be linked to programs built with any C or C++ compiler.

My L4Re port of PThread-Workqueue contains code written in C++ so it can use the more advanced (and more convenient) interfaces of L4Re. However, as C++ is the prevalent language of L4Re, this does not add any extra build requirements.

3.2. Block Literals

Blocks are almost entirely handled by the compiler. They only require a tiny runtime library, which handles management functions such as copying a Block from the stack to the heap (`Block_copy`) where it can outlive the scope it was created in. The little amount of code necessary for these tasks is already written in pure, highly portable C and did not require any adjustments to run on L4Re.

3.3. Grand Central Dispatch

Although the official GCD implementation [Teaa] comes with general support for GNU/Linux (among other platforms), this compatibility was broken by recent up-

dates. Therefore I resorted to a third-party fork of the library [Hut], which already incorporates patches by Mark Heily and Nick Hutchinson, which fix these problems. Thanks to this good starting point only minuscule code changes were necessary and I did not need to spend much time and effort on adjusting the library myself.

3.3.1. On L4Re

Even if all its supporting libraries are available GCD does not work on L4Re without minor modifications because the system is missing features which GCD makes use of. For those parts of the library code that are essential to GCD's core functionality, I created wrappers that emulate the system behavior expected by the library. For example, one such wrapper queries the scheduler of L4Re to learn the number of available CPUs[1]. I replaced all non-essential parts with empty implementations to quench compiler errors. The latter group of replacements consists of functions and header files, which are only used by those parts of GCD that are inoperable anyway due to my deliberately incomplete KQueue port (see section 3.4.2 on page 33). This applies to the majority of GCD's Dispatch Sources and, in extension, to their convenience wrappers, the Dispatch I/O API. However, the core functionality of GCD, that is its Dispatch Queues as well as the related dispatching mechanisms, is fully operational on L4Re.

3.3.2. Uncoupling from Supporting Libraries

Before deciding to use minimally modified versions of the existing GNU/Linux ports (see also section 3.1), I had considered turning GCD into a self-contained library through direct incorporation of its supporting libraries' functionality. KQueue and PThread-Workqueue add considerably to the overall code base of any application built on top of GCD and currently there are no plans to use either as a stand-alone library. Therefore, it seemed to be an interesting option to dispose of them and adapt GCD to run directly on L4Re instead. However, after carefully considering the following pros and cons, I dismissed this idea:

- My investigation of GCD's source code soon revealed a thread pool implementation readily available within the library, which can directly replace PThread-Workqueue. Unfortunately things are completely different in respect to KQueue, which is needed for internal synchronization within GCD and thus cannot be eliminated that easily.

- The coding style of GCD does not invite changes to the library's code at all: Explanatory comments are all but non-existent and often just refer to any entry in Apple's internal bug-tracking system. Furthermore, vital parts of the code make heavy use of the precompiler's `#define` directive (often in multiple layers) to create constructs reminiscent of C++ classes. Performance optimizations are another aspect which makes the code hard to comprehend and to modify.

[1] As a matter of fact, GCD does not require this particular information when it uses PThread-Workqueue. The number of CPUs in the system is relevant, however, if the library's internal thread pool (see also section 3.3.2) is used.

- Changes to any of the involved libraries need to be merged into each new version. Both, PThread-Workqueue and KQueue, were originally kernel interfaces and as such they are not likely to change in the future. GCD, on the other hand, is a user-level library and has already experienced a rapid development since its launch in 2009.
- A strict separation of responsibilities among the three libraries (KQueue for event management, PThread-Workqueue for thread management and GCD for high-level functionality) helps to limit code complexity and facilitates maintenance. Testing, too, can be done separately for each library this way.
- Possible future performance optimization (see section 6.3 on page 82) can directly address the rather small PThread-Workqueue library and does not need to involve GCD at all.
- Although no standalone use for PThread-Workqueue or KQueue is currently planned this is subject to change. In situations where the feature set of one of these basic libraries is sufficient, their independent availability may prove useful.

3.4. KQueue

The original KQueue implementation employs various strategies to achieve the efficiency and scaling properties discussed in section 2.4.3.1: Without doubt, the most important strategy is the use of various in-kernel data structures that handle all bookkeeping related to the event monitoring on behalf of the user space application. As this approach allows the library to keep track of the events the application is interested in, there is no need to provide this information over and over again on every invocation of the library's central `kevent` function. The GNU/Linux version makes use of these central data structures as well, although it does not keep them within the kernel but as part of the user space library.

Another key element to improve scalability is accumulation: The events related to a single monitoring request (e.g., any event concerning a given file descriptor) are not stored as individual items within the data structures I just described. Instead only the first event is recorded in full whereas subsequent occurrences will just update the stored state. The exact semantics of these updates depend on the kind of event monitored. For example, when the user application is interested in reading from a file descriptor, the KQueue library reports the number of bytes that are available for non-blocking read operations. For a timer, however, the number of times it has been triggered would be reported.

Structure of the Library The KQueue library is conceptually centered on event sources and the events they generate. Its implementation builds on three major data structures: the *Filters*, *KNotes* and the eponymous *KQueues*.

The library provides a dedicated Filter for each type of event source it supports. A Filter basically consists of a well-defined set of functions and is used to examine events

from its associated event source. The result of this examination indicates whether a given event is to be reported to the user application. Whenever the application registers a new event for monitoring, it specifies which Filter to use and provides criteria that allow the Filter to identify those events the application is interested in. These criteria are specific to each Filter: Whereas a Filter associated with timer events uses a simple numeric ID for each individual timer, file-related Filters rely on the file descriptor to identify the file in need of monitoring.

A dedicated KNote, a special data structure of the KQueue library, subsequently maintains this event-identifying data as well as the data accumulated from all later occurrences of this specific event. A Filter will create a new KNote for every monitoring request and register it directly with the event source associated to that Filter. It is worth mentioning, that a Filter and its KNotes are highly reminiscent of a C++ class and its instances. The Filter provides the implementations to create/delete/manage its KNotes and to process events from a specific event source. An individual KNote, on the other hand, contains the data related to a specific event the user application wants monitored.

Every time a source generates a new event, all KNotes attached to the the respective source are notified. Each KNote, in turn, activates its respective Filter to check whether the new event should be recorded and to perform the accumulation operation. The first time a given KNote stores an event in this manner it becomes *active*, which means the event represented by the KNote will be reported to the application at the next opportunity (i.e., the next time `kevent` is invoked). By attaching KNotes to predefined hooks of an event source, the KQueue library avoids the polling for new events that would otherwise be necessary and establishes push semantics instead: Each source actively notifies potentially interested KQueues about any event whereupon the source's Filter determines which events should be reported to the user application.

Finally, a KQueue is the central entity that manages all available Filters and forwards monitoring requests from the user application to the appropriate Filter. Furthermore, each KQueue maintains a list of active KNotes (the *active list*), which contains all pending events that have to be reported to the user application.

3.4.1. Active List

The GNU/Linux port of KQueue, which I used as the starting point for my porting work, mainly relies on UNIX domain sockets (`socketpair`) [Gro08] for asynchronous communication between different parts of the library—most notably the active list is implemented in this manner. L4Re, however, does not offer socket support. Consequently, I was forced to adapt the library in this respect and decided to create an implementation that follows the original specification of KQueues [Lem01] more closely than the Linux version.

Aside from this alteration, which the following sections describe in more detail, no major changes to the core parts of the KQueue library were necessary; the KQueue itself, the management of available Filters and, most importantly, the interface towards the user application remain untouched.

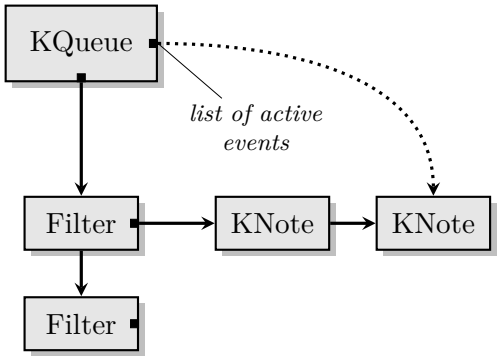


Figure 3.1. *KQueue—Structure on L4Re*

On L4Re In my implementation the active list is a combination of a singly-linked tail queue[2] and a condition variable (see also section 2.2.2.1). This queue, whose head is part of the KQueue data structure, contains all active KNotes. Figure 3.1 shows the overall design of the KQueue library on L4Re. Please note, that the figure does not show event sources because all Filters currently provided by the library work with supplementary, internal events only; see also section 3.4.2 on the facing page

Whenever a Filter determines that an event (received by a specific KNote) is to be reported to the application, it checks whether the KNote is already an element of the active list. If this is not the case, the Filter appends the KNote to the list. If the list was completely empty before, the Filter also signals the conditional variable to unblock a `kevent` invocation that is waiting for new events to arrive. Accordingly, `kevent` blocks on said condition variable (potentially with a timeout) whenever the user application requests to wait for new events and the active list is empty at that moment.

Event Retrieval When a KQueue is about to report pending events to the user application, it appends a special marker KNote to the active list. Processing starts at the head of the list and continues until the marker is encountered; any elements appended after the marker are ignored in the current run.

The extra marker is necessary due to a peculiarity of the KQueue specification: As only the associated Filter has the knowledge required to extract the event-related information from a KNote, it falls within the responsibility of the Filter to process its active KNotes. This detour can be very important as it allows Filters to eliminate outdated events—for example an event related to a file descriptor that was closed after the KNote became active—or merge the latest updates into the event.

However, Filters may not only drop active KNotes in the course of this evaluation process. A Filter may also immediately re-enqueue a KNote to "keep it active". Therefore it is absolutely necessary to mark the original end of the active list before its elements are processed.

[2] A tail queue is a dynamic list whose head stores two pointers; one to the first element and one to the last element of the list. Using these pointers allow for addition or removal of elements at the start and end of the list.

3.4.2. Available Event Sources

As explained previously, event sources need to actively support and interface with KQueue to avoid the inefficient polling for events. Consequently, it is not sufficient to adapt the available Filters to L4Re but their designated event sources, too, would need to be modified to work with the KQueue library. My examination of GCD's code revealed, however, that the library's core functionality actually depends on only two of the events usually provided by KQueue: Timers and User-Defined events. Both are special event types in the sense that they do not monitor existing event sources, but rather establish supplementary, fully user-configurable sources.

As I had not ported the KQueue library to L4Re for its own sake but to lay the foundations for GCD, I decided to only equip it with the Filters associated with these two specific event types for the time being. Most of the remaining event types in KQueue are either not useful on L4Re anyway—for example, POSIX signals are only relevant for better compatibility when porting existing third-party software, which, at this time, rarely (if ever) employs GCD—or were out of the scope of this work due to the necessary adaptations of other system parts that constitute the respective event source.

In section 6.3 on page 82 I will return to this topic and discuss possible future enhancements of the KQueue library whereas the following sections provide a detailed account of how I implemented the User Filter and the Timer Filter on L4Re.

3.4.2.1. User-Defined Events

User-defined events are completely under the control of the application. Instead of being related to an actual event source, User events are triggered manually via normal `kevent` calls to the KQueue library. Each time it triggers a User event, the application must also provide a numeric value, which is stored by the KNote associated with the specific User event. As I explained in section 3.4, all Filters in KQueue support the accumulation of consecutive activation. The User Filter is no exception: Subsequent values will update the one currently stored either by simply overwriting it or, alternatively, via merging of the two values through a binary `AND` or a binary `OR` operation. When the event is reported to the user application, the value which is stored in its KNote at the time will be conveyed as well.

GCD relies on User events for its internal synchronization, which makes this Filter mandatory. Additionally, GCD's Data Dispatch Sources (`DISPATCH_SOURCE_TYPE_DATA_ADD`, `DISPATCH_SOURCE_TYPE_DATA_OR`) are built on and use the respective accumulation options of the Filter.

Implementation User events can only ever be triggered by the application via calls to `kevent`. The processing of these calls is already fully synchronized with all other parts of the KQueue library so that User events require no extra synchronization. This property allows for a straight-forward implementation of the corresponding User Filter on L4Re, which is what allowed me to reuse most of the Filter's code from the GNU/Linux version of the library. The only necessary changes had to be made because of my deviating implementation of the KQueue's active list.

3.4.2.2. Timer Events

The Timer Filter provides an arbitrary number of event sources that, once set up, periodically generate events in user-specified intervals. Subsequent events are accumulated by the Filter through simple addition. In other words: When the user application finally retrieves the pending Timer event, it is informed about the number of Timer ticks that occurred since the last time it retrieved the exact same Timer event.

Events of this type are required for all of GCD's time related functionalities, such as the Timer Dispatch Source (`DISPATCH_SOURCE_TYPE_TIMER`) or delayed dispatching to Dispatch Queues (`dispatch_after`). Although it is, in principle, possible to use GCD even without these features, it would severely reduce the library's usefulness.

Implementation The Timer Filter was more complicated to implement than the User Filter, mainly due to its asynchronous nature in regard to calls from the user application. After its initial setup, a timer generates new events in regular intervals. It goes without saying that the user application must not be blocked during this whole time; the `kevent` invocation that initializes the Timer is completed as soon as the timer is created.

My implementation follows the example of the GNU/Linux version of the KQueue library and uses a dedicated thread for each Timer. In other aspects of the implementation, however, I had to diverge greatly from the original implementation which periodically generates events by blocking on a condition variable (see also section 2.2.2.1) with a timeout that lasts exactly one Timer period. The same variable is also used to cancel the Timer thread whereas Timer events (and the insertion of a timer's KNote into the active list) are implemented with the help of sockets.

As stated at the beginning of this section, L4Re does not support UNIX domain sockets. For this reason, the active list in my implementation is a dynamic list composed of all the KNotes associated with pending events. The head of this list is part of the main KQueue data structure, which in turn is protected by a mutex in order to allow for concurrent, yet consistent modifications by multiple threads. Whenever a Timer thread needs to enqueue the KNote associated with it (because it became active) the thread has to acquire this mutex first. On the other hand, when the user application calls `kevent`, one of the first actions of the library is to acquire this central mutex. In this way it assures that the KQueue it is operating on will not be modified or even destroyed by another thread of the application.

This approach works fine but the use of shared memory for low-overhead communication between the Timer thread and the core library becomes an issue: The Timer thread needs a way to notify its associated KNote of Timer events, which may occur quite frequently, depending on the period the user application chose. Thanks to the KQueue semantics, which I explained previously, it is fortunately not necessary to actually propagate each and every event to the KNote. When pending events are about to be reported to the application, each KNote on the active list is handed to its Filter for extraction of event information. This peculiar design allowed me to have the timer thread handle the accumulation of consecutive events, which boils down to incrementing a counter by one each period. This counter is located in a memory region which is shared by the Timer thread and the core library. It is only updated via atomic operations, that is atomic

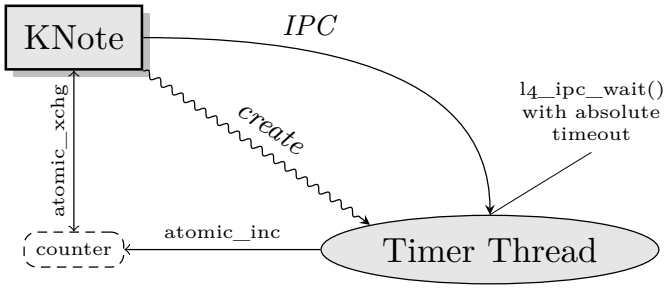


Figure 3.2. *KQueue—Timer*

incrementation by the Timer thread and atomic exchange with zero (i.e., reading and resetting the counter) by the core library. In an effort to avoid setting an arbitrary limit on the number of Timer threads, the shared memory is allocated dynamically (`malloc`).

Under these circumstances a deadlock can arise when the timer thread is about to be terminated. At this point a second mutex comes into play: A Timer thread that wants to enqueue its associated KNote in the active list needs a reference to said KNote and another one to its associated KQueue, which holds the list's head. Furthermore, the timer thread must acquire the KQueue's lock prior to modifying the KQueue or rather the attached active list. It is essential that the reference to a Timer thread's KQueue remains valid as long as the thread exists because the thread may, at any time, try to access said KQueue. Without extra synchronization there is no way to guarantee this property. A simple flag within the shared memory region is not sufficient because checking this flag and accessing its KQueue constitute two separate operations for the Timer thread and the KQueue could be destroyed right between those two actions.

One naive solution is to introduce a second mutex, which must be held by the Timer thread for the whole time is works with its KQueue. This second mutex must also be acquired by the core library before it instructs the Timer thread to terminate. Alas, this approach introduces a circular dependency between the two mutexes[3]—a perfect breeding ground for hard-to-find deadlock errors. Another problem lies in the proper deallocation (`free`) of the shared memory region. The mutex needs to reside within this memory, yet no thread may access the memory (and thus the mutex) after the memory was freed.

In my implementation I obviated this risky situation by replacing the additional mutex with IPC mechanism of L4Re. As this mechanism is inherently synchronous, an IPC operation does not only transfer arbitrary data between two threads but also synchronizes both communication partners. Furthermore, I was able to use the timeout feature of the IPC mechanism as an elegant way to implement the timer's main task—creating accurate timer events. In contrast to condition variables, IPC on L4Re supports absolute timeouts, which, in combination with a high scheduling priority of the timer thread, provide for minimal jitter and drift of consecutive timer events.

[3] As the Timer thread tries to insert its KNote into the active list, it acquires the mutex for the shared memory (to gain safe access to its KQueue) and after that the KQueue's mutex. The core library, which is about to destroy the KQueue, already holds the KQueue's mutex and tries to acquire the shared memory mutex which it needs (as per the described design) to stop the Timer thread.

The final implementation is illustrated in figure 3.2 on the preceding page and works as follows: A Timer thread starts a new cycle by setting up an IPC receive operation (`l4_ipc_wait`) whose absolute timeout is equal to the Timer period. If this operation times out (i.e., one timer period has elapsed) the thread atomically increments the counter, advances the absolute timeout value by exactly one period and the next cycle starts. Should the core library send a new instruction, such as an update to the timer period or a termination command, the IPC will be successful.

When the Timer receives a termination command it frees the shared region of memory and terminates afterwards. Most importantly, after this rendezvous between the core library and the Timer thread, neither of them will access the shared memory again. In addition, the Timer thread will not access any of its external references (its associated KNote and the KQueue) anymore.

3.5. PThread-Workqueue

The PThread-Workqueue library provides a user application with FIFO work queues, which execute the work items dispatched to them in a concurrent manner; see also section 2.4.3.2.

Initially I was determined to make only minimal changes to the code of the GNU/Linux version of the PThread-Workqueue library (see also section 3.1), just as with the libraries discussed in preceding sections of this chapter. Over the course of my work, however, it became apparent that this library would require more extensive modification because L4Re's interfaces are not nearly as complex as those of GNU/Linux. L4Re does neither provide a versatile interface equal to `/proc`, nor does the system's standard scheduler automatically distribute active threads to all available CPUs. Therefore I had to compensate these missing (yet essential) features by expanding the PThread-Workqueue library in terms of load assessment as well as load distribution among CPUs. I will elaborate on these issues in sections 3.5.2 and 3.5.3 on page 38 and on page 43.

In addition to these functional changes there are also more or less "cosmetic" ones, such as renaming and repositioning variables or the addition of in-source comments. Whereas some changes were required for my implementation of a central coordination server—please refer to section 3.5.2.1 on page 39 for details—I made others simply to improve the comprehensibility of the source code.

3.5.1. Worker Thread Pools

The PThread-Workqueue library provides two distinct types of work queues, for which the GNU/Linux implementation maintains two distinct, completely independent worker thread pools:

Normal Work Queues Workers that serve normal work queues are managed by a dedicated thread of the PThread-Workqueue library. This manager runs in fixed intervals and queries the `/proc` pseudo file system for information about the user application as well as other tasks in the system. If all existent workers are busy and the current system load allows for it, the manager creates a new worker thread.

As a special optimization, the last idle worker thread notifies the manager before it starts processing a work item. Provided there are still computing resources available, the manager can thus immediately spawn additional workers.

Excess workers, that is workers which stayed idle for too long, are gradually removed by the manager. For this purpose the manager inserts a special work item (one with a `NULL` pointer in place of a function to execute) in the highest-priority work queue that is currently in use. Whichever worker thread happens to retrieve this item terminates itself. In any event, the manager will always maintain a predefined number of idle workers (*standby workers*) to process new work items without any noticeable delay.

Overcommit Work Queues Overcommit work queues are not subject to any kind of central coordination—in particular, there is no manager involved—which greatly simplifies their worker's life cycle. Each time a new item arrives in an overcommit work queue the library checks for idle overcommit workers. If there is one available it is woken via a dedicated conditional variable; if there is none, a new worker is created instantly.

When an overcommit worker becomes idle, it waits for a very limited amount of time by blocking on the condition variable I mentioned in the previous paragraph. Should new work arrive, the worker is woken up and proceeds to process the new item. Workers that do not get assigned new work before the grace period is over simply terminate themselves.

Such a strict separation of thread pools excludes the possibility of workers that migrate between both pools and, in turn, significantly simplifies thread management. Please also note, that worker termination within both pools is indeterministic—it is left up to chance which excess thread is eliminated. This lack of precise control will become significant when I present my approach for fully utilizing the system's CPUs (see section 3.5.3 on page 43).

Work Queue Priority When I introduced PThread-Workqueue in section 2.4.3.2 I also mentioned that each work queue has a priority that is quite similar to the priority assigned to ordinary threads. The GNU/Linux implementation contains code that allows changes to the scheduler priority of a worker so as to match the priority of the item this worker is currently processing. However, this code lies dormant and a work queue's priority is implemented indirectly instead: Whenever a worker thread retrieves a new item, it scans all active work queues (of the type that the worker services—normal or overcommit) starting from the highest-priority queue and processes the first item it encounters this way. From the scheduler's point of view, though, all workers run at the same priority. Therefore the scheduler is bound to treat all workers the same and assign equal shares of the available compute time to each.

In my L4Re port of PThread-Workqueue I tried to directly expose work queue priorities to the system in such a way that the scheduler prefers high-priority items or rather the worker threads processing these items. Towards this goal I implemented the missing parts of the original library's mechanism of adjusting thread priorities and established

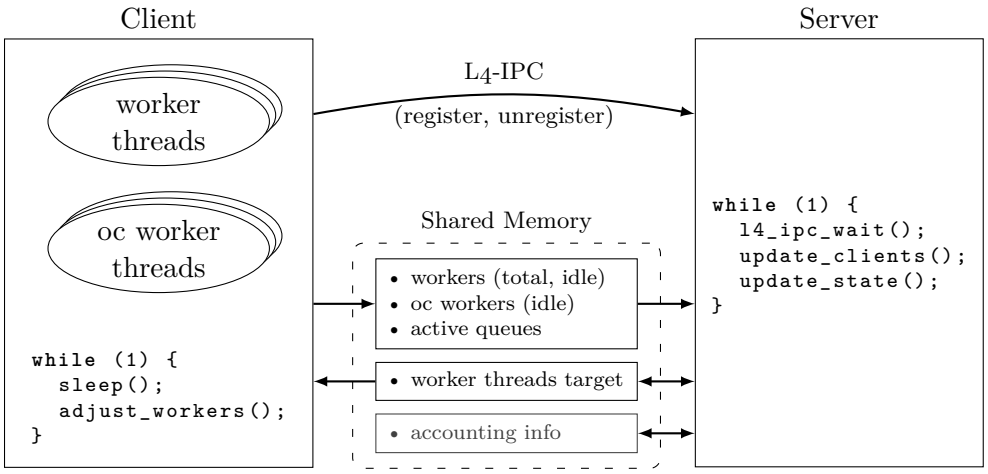


Figure 3.3. *PThread-Workqueue—Server Mode*

a mapping between the four PThread-Workqueue priorities and L4Re's 256 scheduler priorities. Unfortunately, this led to unexpected behavior in my later benchmarks: Some of them would halt at arbitrary points whenever I used worker threads with a priority greater than one (see also section 5.2.1 on page 64). The general problem remained reproducible, however, the actual error occurred at different points during the execution without any obvious pattern. Although I was not able to pinpoint the source of this error precisely, I suspect a conflict with L4Re's PThreads implementation.

3.5.2. Adaptation to System Load

For the thread pool manager to adapt the set of workers to the current system load, the PThread-Workqueue library needs some way to determine the actual load on the system at a given time. The GNU/Linux port of the library, uses the `/proc` pseudo file system to query all relevant parameters—that is the number of active (non-blocked) worker threads of the current task and the total number of active tasks in the system. L4Re, on the other hand, is designed as a decentralized system with strong isolation properties. As such it does not offer an interface comparable to `/proc` nor any other direct way to determine the state (or even the presence) of other user tasks. Therefore, I needed an alternative to assess the current system load.

In the course of my work I developed two possible implementations: a central coordination server and a task-local estimator. Both implementations ultimately rely on the CPU idle time reported by L4Re's scheduler to deduce the current load; please refer to section 3.5.2.3 on page 40 for details of this scheme. In addition, the server maintains communication with all tasks that use PThread-Workqueue and is therefore able to gather detailed runtime data about these tasks. This way the server can built a more precise view of the current load situation than from CPU idle time alone. Furthermore, the server provides coordination among multiple tasks using PThread-Workqueue; for example it may prevent two tasks from concurrently spawning new workers although the current load situation only allows for one.

I started out with the server approach because it appeared promising for its superior information about the system as a whole. However, due to its disadvantages (discussed in section 3.5.2.1), later improvements of the local estimator and its inherent incompatibility with L4Re's principle of decentralization, I abandoned this approach. Nevertheless, the once prevalent server mode of PThread-Workqueue has had a lasting influence on the overall library design. The local estimator was originally intended as a drop-in server replacement; it is easy to setup and thus most suitable for testing scenarios. Reverting the various (structural) changes I had made to the library would have cost too much time without any substantial gain and maybe, in the future, the server approach will even become an appealing option again.

3.5.2.1. Central Server

The main advantage of a central server in comparison with a purely task-local approach (such as the estimator I will present in the following section) is its ability to collect very accurate runtime data from all tasks that rely on PThread-Workqueue for concurrency.

In regular intervals, the server accumulates all data available to it—from its clients as well as from the scheduler—and uses these to establish and maintain an up-to-date view of the system. Please refer to section 3.5.2.3 on the following page for a detailed discussion of the general principles of system load assessment and worker management as these are shared with the local estimator. From this information the server subsequently derives optimal new target values for the number of (normal) worker thread of each client. The server takes full control over this decision, the pool manager of the core library merely creates or terminates worker threads to meet the target set by the server.

Its central position does not only enable the server to base its decisions on global information, but also to coordinate the thread pools of its clients; for example to ensure a fair distribution of workers instead of the "first come, first server" strategy the task-local estimator is forced to use. Figure 3.3 on the preceding page illustrates the overall client–server design I employed for my PThread-Workqueue port on L4Re.

Communication When the user application creates its first work queue, the client registers at the server via IPC and establishes a single page of shared memory, which is subsequently used for transparent, zero-overhead communication. I restructured the code of the PThread-Workqueue library in such a manner that all its internal bookkeeping data—most notably the total number of (normal) workers and how many of those are idle—are placed within this page of memory shared with the server. Another part of the shared page is used by the server as a return channel to each of its clients: The server specifies the target value for the number of (normal) worker threads to establish, which the client acts upon. The remainder of the shared page is used by the server for maintaining client-related data so as to make efficient use of the surplus memory.

The crucial point in this scheme is that both regions, the one written by the client and the one written by the server, are strictly separated so that each piece of data is either written by the client or by server, never by both. This strategy avoids the need for any synchronization because its design precludes concurrent writes to the same location. The specification of the x86 architecture guarantees that read and write operations on the integral types of data used are consistent in any case [Cor13, Section 8.2].

Problems By design the PThread-Workqueue server constitutes a central entity in the system. Not only does this contradict L4Re's general principle of decentralization, it also creates a single point of failure and a potential bottle neck in larger systems.

An even bigger problem are the server's limitations: It is only able to monitor and control applications that rely solely on PThread-Workqueue for task-level concurrency. Even though the server implicitly learns about the system load imposed by traditional threads from the scheduler, it lacks the means to influence these tasks. Then again, the server has no actual control over its own clients either; it has to rely on the full cooperation of all its clients, which may report wrong data or simply ignore the target values set by the server.

3.5.2.2. Local Estimator

Unlike a central server, the task-local estimator has no means at all to directly collect information about the state of other tasks in the system; not even about those that employ PThread-Workqueue. Instead, it relies solely on the scheduler to assess the current system load and, based on this information, decides how many worker threads to employ. In section 3.5.2.3 I will explain both processes in detail.

Originally designed as a server replacement, the local estimator uses the same interface as the server. With both the core library and the estimator running in the same address space shared memory is readily available. The estimator does not run in a thread of its own but is invoked by the thread pool manager at the beginning of its periodic run, mainly to avoid synchronization problems between the two. Just like the server, the local estimator updates its system view and adjusts the worker target value accordingly. The manager, in turn, creates or terminates worker threads to match this target value.

Even though I originally introduced this design to conceal the difference between the server mode and the serverless operation of the PThread-Workqueue library, it also uncouples manager and estimator. This is a welcome side effect because it allows for easier adaptation of both. Over the course of my work, the boundaries between estimator and core library became less defined (compare section 3.5.3.3 on page 45), deprecating the server in turn.

3.5.2.3. System Load Assessment and Worker Management

Even though L4Re does not disclose any information about other tasks, its scheduler can be queried about the idle time of each individual CPU in the system [Drea]. The returned value indicates how many microseconds a given CPU has been idle since the start of the system. With some further processing it is possible to turn this information into a useful indicator of the current system load: Once per *assessment period* (one second in my current implementation), the server or the task-local estimator retrieves the idle times of all available CPUs. It maintains a local copy of the retrieved values and uses them to calculate the idle time of each CPU within the previous assessment period. The three most recent idle-time values for each CPU are then used to estimate the current system load and determine the optimum number of (normal) worker threads.

```
1  if (idle_time[total][current]  > min_idle_time &&
2      idle_time[total][previous] > min_idle_time &&
3      idle_time[total][oldest]   > min_idle_time) {
4
5    /* create extra worker thread */
6  }
7
8  ...
9
10 if (nr_of_threads > nr_of_cpus &&
11     (idle_time[total][current]  +
12      idle_time[total][previous] +
13      idle_time[total][oldest]) < min_idle_time) {
14
15   /* terminate non-idle worker thread */
16 }
```

Listing 3.1. *PThread-Workqueue—Creation/Termination of Extra Worker Threads*

Thrashing Unless care is taken, the dynamic creation and termination of worker threads can easily lead to *thrashing*—new threads are created just to be terminated after a short time with both operations causing extra overhead.

As overcommit workers are not coordinated by a manager but spawned whenever necessary, the only way to reduce thrashing is to delay thread termination. Therefore an overcommit worker enters a grace period after becoming idle and terminates only when no new work is assigned to it during this time; see also section 3.5.1.

The number of normal workers, however, is controlled directly by the pool manager, which creates or terminates threads. In the GNU/Linux version of the PThread-Workqueue library the manager gathers data about the current system load and subsequently uses this information to determine and establish the optimum number of worker threads. In my port this responsibility lies with the central server or the task-local estimator, whichever the user chooses. After updating its internal view of the system the server/estimator determines whether the current number of workers should be adapted and sets an according target value. In doing so, it uses two strategies to avoid thrashing: conservative thread creation and gradual termination of idle threads.

Thread Creation Under normal circumstances my PThread-Workqueue implementation allows one worker per CPU available in the system. If each worker fully utilizes the CPU it is assigned (see also section 3.5.3 on page 43) this number represents the optimum because it prevents useless competition among workers. However, a single worker does not always saturate a CPU. Therefore, the library creates an extra worker thread whenever the *total idle time of the system* (i.e., the accumulated idle time of all CPUs within one assessment period) exceeds a predefined threshold.

In an effort to avoid both, overloading of a CPU and thrashing, I followed a rather conservative approach: Only a single new thread is allowed in each assessment period. In addition, the total idle time of the system must exceed a predefined value (100 ms in my current implementation) for *all three* recorded assessment periods; see also listing 3.1.

The GNU/Linux version of PThread-Workqueue features an interesting optimization: Before the last idle worker starts to process a work item, it notifies the manager, which will immediately check whether to create an additional worker thread. In this way, the manager may react quickly when worker threads are "lost" because of work items that use blocking operations; for example input–output handling or synchronization. However, due to the way system load assessment works on L4Re I was not able to reproduce this desirable behavior. Merely activating the manager early does not suffice because in that case its decision would be based on outdated values for system idle time. Neither is it an option to update these values at arbitrary times because shorter periods between two queries about system idle time inevitably lead to less reliable readings.

The lack of this optimization—in combination with the peculiarities of one of the applications I used for my benchmarks (see section 4.2.1 on page 51)—entailed massive performance losses due to the time that passes before the manager compensates for a blocked worker thread; please refer to section 5.2.3.2 on page 70 for more details.

Thread Termination Idle (normal) worker threads are terminated gradually by "accumulating idle seconds", a technique I adopted from the GNU/Linux version of the library. All idle threads that exceed a predefined upper bound (`idle_threads_threshold`) of standby workers are counted on each assessment period and the results for the different assessment periods are added up. The counter is reset to zero if no idle workers are found or after at least one worker was terminated. Another predefined parameter (`idle_seconds_threshold`) specifies for how long idle workers are kept and once the sum of "idle seconds" exceeds this parameter, the target value for workers is decremented accordingly:

```
if (nr_of_idle_threads > idle_threads_threshold) {
  idle_seconds_accumulated += nr_of_idle_threads;
}
else {
  idle_seconds_accumulated = 0;
}

threads_to_stop = idle_seconds_accumulated /
 ↪ idle_seconds_threshold;

if (threads_to_stop > 0)
  idle_seconds_accumulated = 0;
```

The described scheme ensures that the more idle workers there are, the faster they are cleaned up. On the other hand, it avoids abrupt termination of idle workers and thus helps to smooth the transition from higher workloads to less demanding ones. Whereas both parameters involved in the process are currently constants, future performance optimizations may well apply automatic runtime tuning to them.

If there are more threads than available CPUs, the server/task-local estimator will also consider terminating non-idle workers. This exception is necessary because the CPU load imposed by a worker thread is highly dynamic: Each time the worker starts processing a new work item its load characteristics may change completely. A thread that barely utilized the CPU at all (think of input–output operations or waiting at a mutex) can later execute a work item that performs heavy number crunching. If

there are more worker threads than CPUs and all workers happen to process CPU-intensive work items, threads start to compete for computing time which will likely impede performance.

This termination of extra threads relies on the system idle time, just as their creation does. I applied a hysteresis scheme here, to prevent thrashing: For the creation of an extra thread the total idle time has to exceed a pre-set threshold for all three recorded assessment periods; thread termination, on the other hand, will only commence if the *accumulated* total idle time of the system over all three periods falls below said threshold; see also listing 3.1 on page 41.

3.5.3. Utilization of Available CPUs

As a microkernel, Fiasco.OC provides only mechanisms without enforcing a particular policy and L4Re's basic system services try to maintain this property (see also section 2.1). Consequently its scheduler is kept very simple and, in particular, does not distribute threads among the CPUs in the system. The scheduler simply runs any thread on the first available CPU unless it is explicitly instructed to do otherwise. Linux' scheduler, in contrast, handles this distribution on its own. Therefore, the GNU/Linux version of the PThread-Workqueue library merely has to maintain an appropriate number of worker threads in order to fully utilize all of the system's CPUs.

On L4Re I had to implement a mechanism that explicitly assigns each worker to a CPU. As stated in section 3.1, my work is focused on providing GCD (which requires PThread-Workqueue) on L4Re, not on optimization. For this reason, I started with a simple round-robin scheme, which I gradually enhanced when my preliminary tests exposed severe shortcomings. I never based my calculations on the actual CPU load of a worker, however, because the computational demands of a worker may change each time it starts to process another work item. Instead I used the number of workers assigned to each CPU. The assignment of workers to CPUs is permanent (unless workers may migrate between CPUs; see also section 3.5.3.3 on page 45) so any equal number of workers on each CPU will lead to a uniform distribution of computational load in the long run.

3.5.3.1. Round Robin

In my first approach both normal workers and overcommit workers were assigned to the available CPUs in a simple round-robin fashion. However, tests soon showed that this strategy may lead to an imbalance when certain CPUs are loaded with many normal workers while others have only overcommit workers assigned to them. Usually overcommit work queues are used for short, but important tasks, whereas normal work queues handle CPU-intensive tasks. At least, this is the use intended by the PThread-Workqueue library.

In an effort to counter this imbalance I enhanced the simple scheme to handle CPU assignment independently for both types of workers—still in a round-robin manner though—when yet another problem arose: Workers are terminated in an uncoordinated way. Overcommit workers terminate themselves if they stay idle for too long and

```
1  void assign_thread(pthread_t tid, bool is_oc) {
2    static int threads_on_cpu[2][MAX_CPUS];
3    static int next[2];
4
5    const int cpu_start = atomic_inc(&cpu_next[is_oc]) % MAX_CPUS;
6    int min = 2000000; // arbitrary large number
7    int use, cur = cpu_start;
8
9    do {
10     if (threads_on_cpu[is_oc][cur] < min) {
11       min = threads_on_cpu[is_oc][cur];
12       use = cur;
13     }
14
15     cur = (cur + 1) % MAX_CPUS;
16   } while (cur != cpu_start);
17
18   /* assign thread 'tid' to CPU 'use' */
19
20   atomic_inc(&threads_on_cpu[is_oc][use]);
21 }
```

Listing 3.2. *PThread-Workqueue—Assigning a CPU to a Newly Created Worker*

although normal workers are coordinated by a manager their termination is not (see also section 3.5.1). As a consequence, it is possible that, by chance, all workers assigned to a certain CPU are terminated. This CPU would then be completely idle, while others might be overloaded. Even worse, newly spawned workers are not necessarily assigned to the idle CPU either.

3.5.3.2. Balancing

In an effort to ensure a more favorable distribution of workers among all available CPUs, I introduced a more elaborate mechanism that replaced the simple round-robin allocation. The library keeps track of the number of workers assigned to each CPU—once again, separately for normal and overcommit workers. As these counters are only updated by atomic instructions, there is no need for extra synchronization of access to the data[4]. With this information, each spawned thread is assigned to the CPU that (at that time) hosts the smallest number of worker of the respective type.

Again, the basic version of the algorithm causes a problem. When multiple CPUs host the same number of workers and several new workers are spawned within a short interval of time, the per-CPU thread counters are not updated in time and all the workers are assigned to the same CPU. This situation is quite common for overcommit workers but it also occurs for normal workers when the library starts up and creates a minimal set of workers for standby.

[4] Read-only operations are unproblematic because the x86 architecture itself guarantees their atomic execution [Cor13, Section 8.2].

I solved this problem by incorporating the round-robin scheme: The enhanced algorithm still uses a pair of counters—one for normal workers, one for overcommit workers—for each CPU in the system to record how many workers of each type are hosted by the respective CPU. In addition, the library keeps track of the last CPU that received a new worker; this too is done independently for both types of workers. Now the search for the CPU hosting the least number of workers starts with the CPU following the one that received the last worker and another CPU is only preferred if it hosts fewer workers.

Listing 3.2 on the facing page shows a simplified pseudo code implementation of the final algorithm. Please take note that the values of `threads_on_cpu` may change while the loop is active. I accepted this inaccuracy as well as the eventual overflow of `cpu_next` because the uncoordinated termination of worker threads turns the whole algorithm into an approximation anyway.

3.5.3.3. Migration

Finally, I experimented with simple thread migration between CPUs, both to (indirectly) account for the differences in the CPU utilization of workers and to rebalance the load when workers were terminated and no new ones are being created.

As discussed in section 3.5.2.2 the local estimator uses the idle times of all CPUs in order to assess the overall load of the system. For this purpose it maintains a complete list of the idle time for each single CPU over the course of the last three seconds. I extended the estimator in such a way, that it also determines the CPU with the greatest amount of free capacity by calculating a weighted sum of the idle times of each CPU, whereby more weight is placed on more recent values than on older ones:

```
weighted_idle_time[cpu] = 3 * idle_time[cpu][current]  +
                          2 * idle_time[cpu][previous] +
                          1 * idle_time[cpu][oldest];
```

A variable (`most_idle_cpu`) with global scope holds the final result: the ID of the CPU with the most idle time. After retrieving a new work item to process, each worker checks whether it already runs on said CPU and tries to migrate there if that is not the case. Obviously, this naive implementation would cause *all* workers (that start processing a work item) to jump to the chosen CPU and, as a result, overload that CPU. I countered this problem by introducing a scheme where a worker has to claim the idle CPU by setting the global variable to a predefined marker value (`-1` in my implementation). Thanks to atomic instructions, which ensure data integrity in the presence of concurrent access, only a single worker can successfully claim the idle CPU and migrate to it:

```
int best_cpu = most_idle_cpu;

// check + try to claim CPU
if (best_cpu >= 0 &&
    compare_exchange(&most_idle_cpu, best_cpu, -1)) {

  // success -- now switch to the new CPU
  migrate_to(best_cpu);
}
```

Whereby `compare_exchange` is a compare-and-swap function that atomically

1. checks whether `most_idle_cpu` still holds the value that was previously stored in `best_cpu`,
2. writes `-1` as the new value of `most_idle_cpu` (thereby claiming the idle CPU for the current worker thread), provided (1) is the case, and
3. finally returns `true`.

Should the check in (1) fail, the function will simply return. This indicates that the CPU has already been claimed by another worker.

Preliminary tests showed, however, that due to the simple design of my migration algorithm a single worker will often "bounce" between two CPUs—switching to an idle CPU and going back after processing one work item because the original CPU had become idle. Furthermore, benchmarks (presented in section 5.2.3.3 on page 75) did not indicate any significant overall performance impacts, positive or negative, which is why I decided to disable thread migration for the time being. However, in retrospect my benchmark workloads turned out to be rather special (see section 5.2.4 on page 78) so the results may well differ with other applications.

4. Remodeling

In order to gauge the overall performance of GCD and the performance of my L4Re port in particular, I remodeled two multi-threaded applications to use GCD instead of PThreads. Whereas I discuss the results of my comparative runtime benchmarks with these applications in section 5.2 on page 64, this chapter addresses the preceding restructuring process.

After explaining my criteria for the selection of a suitable test application, I will briefly introduce the chosen programs. To avoid repetition I refrained from describing my work on each application in detail and instead discuss only those parts of particular interest—problems that occurred in the process and strategies that proved to be useful.

4.1. Selecting an Application

My intention in remodeling an existing application to evaluate the performance of GCD was to test the real-life performance impact of using the library. Whereas specifically created benchmarks (or even micro-benchmarks) are perfect to analyze isolated properties of an implementation, instrumenting a complex real-life application gives a better impression of the practical implications. Furthermore, I needed an implementation to test GCD against. This implementation should be based on one of the competing technologies for expressing task-level concurrency (see also section 2.2). I deemed PThreads, the prevalent concurrency library, the best choice for this purpose as I expected to find a variety of applications from which I could choose for benchmarking.

4.1.1. Requirements

Finding a suitable application for the intended benchmarks actually proved to be difficult because there are many and sometimes contradicting requirements. In particular, the application in question ...

- needs to possess inherent parallelism. Multi-threading cannot improve the performance of programs with little to no concurrency. This situation may well be interpreted as a rather extreme case of Amdahl's Law [Amd67].

- must be complex enough to provide realistic results. As I explained previously, only a complex real-life application can give an impression of real-life performance. On the other hand, a code base that is too large would have made the remodeling process impossible in the limited amount of time scheduled for my work.

- should exhibit a dynamic level of concurrency. An application is dynamic in this sense when the extent of concurrency varies during its runtime: Phases with few activities running in parallel alternate with phases of more parallelism. Such dynamics allow assessing the usefulness of GCD's automatic adaptation to system and application load as well as the suitability of my current implementation of the mechanism (see also sections 2.4 and 3.5.2).

- must have its source code published and preferably use a permissive license that permits modifications to this code. The availability of the source code is crucial for remodeling and adapting the application in the first place, whereas a suitable license allows me to publish these modifications so that others can reproduce my benchmark results.

- should be already available on L4Re or at least be easily portable to the system. Any faithful comparison of the original application and its GCD-based version require both versions to run on the same platform; hardware and software. The main intent of this experiment is a performance evaluation of my GCD with L4Re as the software platform. Thanks to L4Re's high degree of POSIX compliance this requirement is not difficult to meet though.

Initially, I considered database management systems a promising field—especially when taking into account that *SQLite* is readily available on L4Re. It soon became clear though, that SQLite does not employ threads in its internal design. Although the library is thread-safe, there is no inherent concurrency and the only source of parallelism is the application built on SQLite. It was not my intention to create a new application for my benchmarks, as I explained at the beginning of this section, so I discarded this option.

Another idea, the remodeling of a network stack, also turned out to be a dead end: *Lightweight IP* (lwIP) had previously been ported to L4Re and looked very promising because it met my requirements. However, the version available was outdated and had exhibited unpredictable timing behavior in previous experiments. Updating the stack to a newer version could probably have fixed these issues but was outside the scope of my work. Therefore I decided not to use lwIP either in order to avoid problems with inaccurate measurements later on.

4.1.2. Fraktal

In absence of a suitable application (and also to get better acquainted with GCD) I decided to remodel a simpler application first and found one among the sample programs of L4Re: Fraktal is a multi-threaded tool that was written using only the native interface of L4Re. It calculates the Mandelbrot set and, optionally, displays a graphical representation thereof. Except for its small code base, Fraktal meets the previously described requirements. The application distributes its computational load across a set of worker threads by splitting the domain of definition into equal-sized *slices*. A dedicated manager thread coordinates this distribution: It prepares all slices and assigns them to workers in a round-robin fashion, waiting when no idle workers are available.

Once all slices have been assigned, the manager waits until all workers are finished. The IPC mechanism of L4Re is employed for all synchronization and each worker shares a region of memory with the manger for data exchange. Please note that there is no dependency among the workers because each of them only interacts directly with the manager.

Numerous parameters (such as the number of slices and threads) are adjustable by the user at runtime and offer a wide variety of benchmark scenarios. Finally, the computation time required by different segments varies due to the definition of the Mandelbrot set[1] and thus the application displays dynamic levels of concurrency.

Adaptation to Grand Central Dispatch Thanks to its rather simple functionality and small code base the conversion of Fraktal from PThreads to GCD its underlying concurrency library was quickly done. Due to the regular structure of its primary algorithm, in particular, I was able to redesign Fraktal from scratch and fully adapt it to the GCD concept of Dispatch Queues. Basically, each slice is processed by a dedicated Block which, in turn, is dispatched to a Concurrent Queue (see also sections 2.3 and 2.4.1). Please refer to section 4.2 for a detailed discussion of other useful strategies I devised during the remodeling process.

4.1.3. Stockfish

The fruitless search for a more complex application to benchmark finally came to an end when I started to consider games as an option and encountered Stockfish. In order to discuss Stockfish—an advanced open source chess engine [RKC]—I will need to use terminology related to *artificial intelligence* (AI) in general and chess programming in particular. A thorough introduction to this vast field would be completely outside the scope of this work, so please refer to sources such as [LC] for detailed explanations of unfamiliar terms.

Basically, Stockfish builds upon the well-established Minimax algorithm, which is a standard depth-first search for zero-sum games that is often used in AI. The algorithm is further improved through a multitude of enhancements and optimizations—from alpha–beta pruning and iterative deepening to dedicated strategies for opening, midgame and endgame situations. In addition to these theoretical improvements the implementation employs numerous performance optimizations for PThreads. Most notably, Stockfish supports load balancing with up to 64 worker threads and makes extensive use of various global and thread-local hash tables that cache previously encountered game positions, thus avoiding their repeated evaluation. The exact number of workers that is used is chosen according to the number of CPUs available in the system but can be adjusted by the user.

A new search always starts off with a single thread and is then *split* up at a particular point of the search algorithm if specific conditions are met. One such condition is the

[1] The Mandelbrot set is defined as the "set of complex c-values for which the orbit of 0 does not escape under iteration of $x^2 + c$" [Dev]. In order to determine whether a given point of the domain is an element of the Mandelbrot set, Fraktal uses an iterative algorithm. The sooner the orbit of 0 does escape for a point under consideration, the sooner this algorithm terminates.

availability of idle worker threads; another is that the search has reached a depth of the search tree at which parallelization will actually improve performance. Each time the search is split in this manner it incorporates all idle workers[2] and creates a new data structure (*Splitpoint*) which is subsequently used for communication among all threads that are involved in this split. In order to prevent data corruption, the Splitpoint is protected with a mutex. It is of particular importance, that the search process is designed in a way that barely involves any waiting. This design allows for high CPU utilization.

Adaptation to Grand Central Dispatch Adapting Stockfish to use GCD instead of PThreads was much harder than remodeling Fraktal had been. Despite its extensive documentation—including source code comments as well as detailed descriptions of the used algorithms—the complex, highly-optimized structure of Stockfish makes it difficult to comprehend. Due to its multitude of optimizations, in particular, even small changes may have tremendous consequences in terms of correctness and performance.

Whereas I was initially determined to restructure Stockfish from scratch as I had done with Fraktal, I soon had to give up on these ambitions. As I lacked both time and expertise in the field of AI programming, I was unable to adapt the used algorithms and limited myself to modifications of the implementation as such. Even though I finally managed to remodel the application to run on top of GCD, it became apparent during the process that Stockfish's basic algorithm (Minimax, a depth-first search) and the mechanisms that GCD offers are incompatible in many ways. GCD's central concept of a FIFO queue is perfectly fit for the implementation of a breadth-first search yet unsuitable for a depth-first approach. Furthermore, the required adaptations disable important performance optimizations; in particular the previously described thread-local caches are reduced to being short-lived and Block-local, which almost completely defeats their purpose. In section 4.3 on page 58 I will elaborate these problems further and discuss the way in which I managed to diminish these negative effects of the remodeling process.

Basically, a Concurrent Dispatch Queue and the synchronization mechanism of GCD (see section 2.4.2) are used to replace the workers threads of the PThreads implementation. While the Splitpoint data structure remained in place, I was able to significantly reduce its size and complexity thanks to the variable capture properties of Blocks (see section 2.3.2). Unfortunately, splits of the search do not correspond to the levels of the search tree in Stockfish's design and there is no way to determine the optimal number of Blocks to generate for a split. Please keep in mind that in the original PThreads implementation the number of available workers limits this *split factor*. Each time the search splits a fixed number of Blocks is created instead. Over the course of my later benchmark tests (presented in section 5.2.3.2 on page 70) I evaluated how this number affects the overall performance of the GCD version of Stockfish.

[2] As a matter of fact not *all* idle workers can be employed because Stockfish uses the *helpful master* extension of the *Young Brothers Wait Concept*.

4.2. Strategies

The paradigm of GCD differs fundamentally from the traditional use of threads and this complicates the remodeling of existing applications; see also section 4.3 on page 58.

In the following sections I will present a number of general strategies I developed during my work on Fraktal and Stockfish and which proved useful for this process. Please note, that I omit the mandatory error checking in code samples presented to keep them short and concise.

4.2.1. Fork–Join Structure

Often there are situations where just a small part of an algorithm may run in parallel. The results produced by this part, however, are required before the algorithm can continue its execution.

A combination of a Concurrent Dispatch Queue and a dedicated Dispatch Group (see also sections 2.4.1 and 2.4.2) are the best way to express this scheme with the mechanisms of GCD. All concurrently executing items—either Blocks or ordinary functions—are dispatched to the Concurrent Queue in an asynchronous manner and at the same time become members of the Group (`dispatch_group_async`). This approach ensures a maximum of concurrency as well as an easy way to continue once all items in the Group have finished their work. The optimal solution in full compliance with GCD's paradigm of asynchronous dispatching would have the Group release a finalizer Block once all its members are done. In this case, there occurs no blocking at all, which benefits overall performance.

However, existing applications that are built on PThreads are usually designed to avoid concurrency and therefore do not allow for the use of this optimal design; a problem I will discuss in section 4.3.1. When remodeling such an application the only possible choice is to wait until all members of the Group complete (`dispatch_group_wait`). It would require a complete redesign of the application to fully adapt it to GCD. Unfortunately, as long as the queue item waits for the Group (or rather for its members) to finish, the worker thread running this item remains blocked.

Even though the underlying PThread-Workqueue library will sooner or later compensate for the blocked worker by spawning a new one (provided the current system load allows an additional thread[3]), additional threads mean additional management overhead and have a negative impact on the overall performance of the application. This effect is aggravated by the L4Re port of the library because the implementation lacks an optimization (see section 3.5.2.3) that significantly reduces the time until a new worker becomes available.

In the remodeled version of Stockfish I tried to reduce the described effect. When the search splits up, a fixed number of Blocks is generated and all of them—except for one—are handled according to the scheme presented in this section: The Blocks are asynchronously dispatched to a Concurrent Queue and a Group is used to keep track of their status of completion. The last remaining Block, however, is invoked directly,

[3] If the system is too busy to allow for the creation of a new worker, no harm is done either as all CPUs are fully utilized anyway.

delaying the moment when the worker thread is blocked to the time after this Block has been executed. In the best case, all other Blocks are already complete at this point so blocking is avoided altogether.

4.2.2. Loops

Standard `for` loops can be unrolled conveniently with GCD's `dispatch_apply` function, which basically dispatches a new item for every iteration of the original loop. However, not all loop constructs can be parallelized so easily.

4.2.2.1. Input Loop

Often, an application employs a dedicated thread for the sole purpose of input processing and the respective thread runs an infinite loop in which it waits for and handles user input. The most intuitive approach to recreate this behavior with the mechanisms of GCD is a Dispatch Source that monitors the standard input file descriptor. Unfortunately, my current L4Re port of KQueue does not feature the Filter required for Sources of this type (see also section 3.4.2) which ruled this solution out. As I did not want to resort to using traditional PThreads either, I had to come up with an alternative.

Dedicated Serial Dispatch Queue For Stockfish this alternative was a dedicated Serial Dispatch Queue on which one single item (containing the original infinite loop) runs forever. Due to the properties of Serial Queues or rather the overcommit work queues they are built on (see also section 2.4.3.2) this approach admittedly results in the creation of a dedicated thread. Yet it fits better into the general design of an application built on GCD.

Self-Enqueueing Block As a variation, it is also possible get rid of the infinite loop and create a Block instead that contains the original loop's body and subsequently dispatches itself again. Optionally, a short delay may be introduced (`dispatch_after`), which reduces the resulting load when the Block does not need to run continuously as is the case for input processing. Please see listing 4.1 on the next page for a code sample. It is important that the Block reference which holds the self-enqueueing Block is declared with either the `__block` or the `static` modifier so that the variable is captured by reference. Otherwise the Block would capture an uninitialized version of the reference, the dispatching of which leads to undefined behavior.

The main advantage of this approach is that the Serial Dispatch Queue may be used for other Blocks as well and I initially applied it in the remodeling of Fraktal as well. The application can handle user input via JDB, the kernel debugger of Fiasco.OC (`l4kd_inchar`), however, this function is as low-level as it can be and always returns immediately, with the return value indicating whether or not input was available. Therefore I used the delayed re-enqueue operation (`dispatch_after`) to avoid behavior similar to busy waiting.

```c
// run by dedicated thread
while (1) {
  /* do work */
}
```

(a) PThreads

```c
__block void(^block)(void);

block = ^{
  /* do work */

  dispatch_after(/* when */,
                 /* queue */,
                 block);
};
dispatch_async(/* queue */,
               block);
```

(b) GCD

Listing 4.1. *Input Processing Loop*

Timer Dispatch Source Later, however, I came up with a third, more elegant solution than re-equeueing Blocks as I just described: A Timer Dispatch Source dispatches a Block, which contains the original (infinite) loop's body, to an arbitrary Dispatch Queue in regular, short intervals. So in the case of remodeling the JDB input processing of Fraktal, the Block checks for new input and processes it, if there is any. For the sake of simplicity I used one of GCD's standard Queues as target for the Timer.

When using this approach, it is important that the processing of each Block must not take longer than the Timer period. If it does, more and more Blocks will accumulate. In this case, it is advisable to use the previously described technique of a re-enqueuing Block because it guarantees that a new instance of the Block is only dispatched after the current one finished. However, as the processing of input from JDB is fast this was not an issue with Fraktal.

4.2.2.2. Enclosing `while` Loop

There are situations where a `for` loop (or any other construct with inherent concurrency) is contained within a `while` loop. In contrast to `for` loops, a `while` loop cannot be unrolled easily because the exit condition usually depends on the loop's body. Therefore, the body must be completely processed before it is known whether the loop will run another time. A straightforward way to express this dependency in GCD would be the one I described in section 4.2.1: Create a new Dispatch Group and use it to wait for the parallel section within the `while` loop to complete and then resume serial execution.

There is a better solution, though, in case the function that contains the outer `while` loop (`func` in listing 4.2 on the following page) runs asynchronously, that is it may return without providing a result while the actual processing is still in progress. Once the function finishes the (out-of-line) processing, it dispatches a Block that signals completion, communicates the final result or does whatever the developers deems appropriate. This completely asynchronous approach is how GCD is designed to work.

Now, provided the function itself is asynchronous, the outer `while` loop can be split up into two asynchronous parts as well. In this particular case the parallelization of the inner `for` works without waiting: Once again, the key is a Block that re-enqueues itself,

```
1  void func(void) {
2    /* preprocessing 1 */
3
4    while (condition) {
5      /* preprocessing 2 */
6
7      for (...) { /* for body */ }
8
9      /* postprocessing 2 */
10   }
11
12   /* postprocessing 1 */
13 }
```

(a) *PThreads—Original Loop with Latent Inner Concurrency*

```
1  void func(void) {
2    static dispatch_queue_t queue;
3    static dispatch_group_t group;
4    static void(^block)(void);
5
6    queue = dispatch_queue_create("", DISPATCH_QUEUE_CONCURRENT);
7    group = dispatch_group_create();
8
9    /* preprocessing 1 */
10
11   block = ^{
12     // stop re-enqueue cycle
13     if (!condition) {
14       /* postprocessing 1 (should signal result) */
15
16       dispatch_release(group);
17       return;
18     }
19
20     /* preprocessing 2 */
21
22     for (...) {
23       dispatch_group_async(group, queue, ^{ /* for body */ });
24     }
25
26     // exeutes after all Group members complete
27     dispatch_group_notify(group, queue, ^{
28       /* postprocessing 2 */
29
30       dispatch_async(queue, block); // re-enqueue the Block
31     });
32   };
33   dispatch_async(queue, block); // start the process
34 }
```

(b) *GCD—Exposing Inner Concurrency*

Listing 4.2. *Parallelization of a Section Within in a* `while` *Loop*

```
1  class Data {
2    ...
3    public:
4    dispatch_queue_t q_access;
5
6    Data(...) {
7      ...
8      q_access = dispatch_queue_create("", DISPATCH_QUEUE_SERIAL);
9    }
10
11   ~Data() {
12     ...
13     dispatch_release(q_access);
14   }
15 };
16
17 ...
18 __block Data d;
19 dispatch_sync(d.q_access, ^{ /* access 'd' */ });
```

Listing 4.3. *Synchronization of Operations on an Object*

but this time it also needs to check the exit condition of the `while` loop in order to break the otherwise endless cycle at some point. Listing 4.2 on the preceding page shows a generalized example—please pay particular attention to the negated exit condition at the start of the Block and also to the fact that the original construct's overall structure is "turned upside down".

The same basic strategy can theoretically be employed for nested loops of arbitrary depth but the resulting design will quickly become overly complicated and difficult to comprehend. Also, the restriction I stated at the beginning of this section still applies. The function containing the original loop(s) must be allowed to return before it final result is available. In other words, it must run asynchronously. Coincidentally, this was the reason why I was not able to apply this technique during my work on Stockfish in the end.

4.2.3. Synchronization

In section 2.4.2 I introduced some common schemes for replacing PThreads synchronization mechanisms with GCD constructs. I will not reiterate those here but present higher-level strategies that built on and extend these basic mechanisms.

4.2.3.1. Serializing Operations on an Object

Stockfish is written entirely in C++ and for this reason I frequently had to protect access to objects shared by multiple threads (or rather Blocks). In such a situation it is advisable to add a new `public` member variable to the respective class that holds a dedicated Serial Dispatch Queue, which will subsequently be used for any concurrent access to the object. Please refer to listing 4.3 for a code sample.

As an alternative option I also considered a completely transparent encapsulation of access to member variables: Here the dedicated Serial Dispatch Queue is introduced as a `private` member variable and thread-safe access methods are added to the class. These methods use the Serial Queue to serialize access to the actual data of the object. Although this approach seems more elegant at first glance because it conceals all synchronization within the object, it has one major flaw: There is no way to access multiple member variables atomically, except through additional methods for this exact purpose. Even if such methods exist, it remains impossible to retrieve a value from the object, process it and write the results back to the object in one single atomic transaction. Furthermore, any access to the object will go through its Serial Queue, even if the object instance is used exclusively by a single thread or Block.

4.2.3.2. Write Access to Captured Local Variables

One of the most intriguing properties of Blocks is their ability to capture all variables in scope at the time of the Block's definition. However, as stated in section 2.3.2, captured local variables are read-only within the body of the Block.

When reading shared data, a copy is usually stored in a local variable for further processing or to avoid the extra synchronization overhead cause by accessing the shared version when the data is needed again. With GCD, a dedicated Serial Dispatch Queue is typically used to implement safe concurrent access to shared data (see also section 2.4.2). But the Block that is being dispatched to this queue to access and retrieve the data cannot write the captured local variables of the environment it was created in.

This problem is typically solved by using a `__block` variable instead of an ordinary local variable. However, a variable with block storage duration is *always* accessed via a pointer indirection [Teab] which may affect the application's overall performance; especially if the technique is used often. Furthermore it involves more modifications to the application that is being remodeled. The situation is reminiscent of a C function whose parameters are passed by value. Modifications made by the function are not visible to its caller. Pointers are used to achieve call-by-reference semantics instead and I adopted this scheme to grant Blocks write access to captured local variables.

Whenever a Block required write access to a local variable, I created a pointer so that the Block may write the original local variable via this pointer. Although the pointer itself is still captured by value, the variable it references may be written.

```
void func(void) {
  int local = 23;
  ...
  int* local_p = &local;
  dispatch_sync(q_mutex, ^{ *local_p = 42; });
  // local == 42
}
```

Please keep in mind though, that the variable which is accessed via the newly created pointer is located on the stack of the current function. While the pointer, which is captured by the Block will be conserved for the lifetime of this Block, the local variable it references will be destroyed along with the stack when the current function returns.

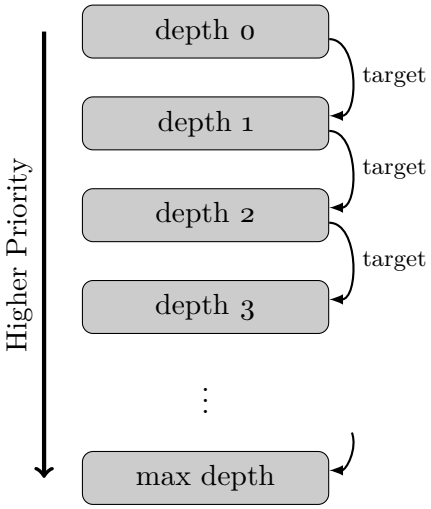


Figure 4.1. *Hierarchy of Dispatch Queues*

Therefore, the discussed scheme will only work reliably when the Block is dispatched synchronously. On the other hand, this is normally case in the described scenario of access serialization.

4.2.4. Stack/LIFO Queue

Even though GCD offers various methods to dispatch items to one of its Queues, there is no way to exert full control over this process or modify the item once it has been placed in a queue. Neither is it possible to remove items from a queue nor does the library support the insertion of items at positions other than the end of a queue. These restrictions become a problem when implementing techniques that are incompatible with the simple FIFO approach of GCD.

I had to deal with this issue when I remodeled Stockfish and realized that using a FIFO queue for parallel execution of different parts of the search process effectively transformed the original depth-first search into a breadth-first search. Although this (unintended) modification of Stockfish's core algorithm does not affect the correctness of the application, it severely impairs several performance optimizations, the most significant of which is alpha–beta pruning.

Later on, I came upon an interesting approach to emulate the insertion of items at the beginning of a queue by using a combination of two Dispatch Queues (one being the target of the other) and queue suspension [Ste11, "Jumping the Queue"]. Building on this basic idea I devised a technique that would allow me to maintain the properties of Stockfish's original depth-first search: A hierarchy of Dispatch Queues with one dedicated queue for each depth level of the search tree. The target of each of these queues is the queue corresponding to the next-deeper level; please refer to figure 4.1 for an illustration.

When the search process reaches a new level of depth (`depth`) and the search is split up, new Blocks are dispatched to the queue associated with the level of depth just entered (`q_work[depth]`). At the same time, the queue associated with the previous, lower level

(`q_work[depth-1]`) is suspended and will stop processing items[4]. Therefore, only Blocks working on the current, highest search depth are still being processed. Once all these Blocks have been completed, the suspension is lifted and Blocks working on the next-lower level of the search tree are processed once again.

Even though this approach did not work out with Stockfish (as I will elaborate in section 4.3.2 on the next page) it can still be useful for similar problems or, with little adaptation, provide a priority queue with an arbitrary number of priorities.

4.3. Problems And Lessons Learned

Remodeling Fraktal was the first task during which I actually worked with GCD on a deeper level. Therefore it took me some time to become acquainted with the library, for example, when to release manually created Dispatch Queues or how to make optimal use of the library's synchronization mechanisms. The simple structure and small code base of Fraktal made the application relatively easy to understand and remodel.

Stockfish, on the other hand, is in a completely different league, both in terms of complexity and code size. This statement does not only refer to the inherent complexity of the search algorithm used by Stockfish, but also to the layers of performance optimizations employed by the application. In the remainder of this chapter I will discuss major obstacles I encountered during my remodeling efforts and try generalize them.

4.3.1. Concurrency Avoidance

Applications that are based on traditional threads usually avoid concurrency whenever possible because threads are rather tedious to handle and introduce unpredictable timing behavior into the application. What is more, threads need to be managed on top of the actual task the application is supposed to accomplish. For these reasons, concurrency is employed sparsely with traditional threads: The majority of the code base is sequential code and only selected parts are parallelized to enable a better performance on modern systems. Usually the results produced by the threads in these parallel parts are subsequently collected and the application goes back to serial execution. Although this structure is reminiscent of Open MP (see section 2.2.2.3) it is also very common in applications that use other libraries to manage traditional threads.

GCD, on the other hand, propagates the opposite approach and the library's motto ("Islands of Serialisation in a Sea of Concurrency") was chosen with good reason: With GCD concurrency is more convenient to handle and becomes the standard case whereas serialization is only enforced when this is strictly required to ensure correct results. In other words, whatever may run in parallel should run in parallel.

For this reason most PThreads-based applications require a full redesign (which may even extend to the underlying algorithms) if they are to be fully adapted to GCD. While I managed to accomplish this task with the rather small Fraktal, Stockfish proved to be too big and complex to make changes of this magnitude.

[4] Please note, that at this point all queues with lower levels (`q_work[depth-2]` to `q_work[0]`) have already been suspended because a new search starts at depth level zero and only gradually enters higher levels of the search tree.

4.3.2. Simplicity of Dispatch Queues

GCD strives to simplify concurrent programming—the whole library is centered around the concept of versatile Dispatch Queues (introduced in section 2.4.1). While this approach allows for a convenient parallelization of applications, it also limits developers in their choice of algorithms or at least complicates the use of schemes that do not fit easily into GCD's FIFO queue scheme.

During my work on Stockfish, for example, the limited interface of GCD forced me to change the semantics of the original code by turning a depth-first search into a breadth-first search. Concurrent Dispatch Queues are the main mechanism offered by the library to enable parallel execution of multiple parts of an application. Stockfish recursively splits up its search process; in the context of GCD this results in the creation of new Blocks. Yet, as a new Block can only be inserted at the end of a queue, all the Blocks in the queue are bound to have run before the new ones are processed. These older Blocks, however, represent game positions from further up in the search tree and therefore the result is a breadth-first search. In fact, a FIFO queue is a typical implementation of a breadth-first search whereas a depth-first search would use a stack or a LIFO queue instead.

Changing these basic principles of the search algorithm defeats many of Stockfish's performance optimizations such as alpha–beta pruning. Unfortunately, I did not fully realized this problem until my benchmarks showed unexpectedly low results for the restructured version of Stockfish while the results for Fraktal remained similar to those achieved the the original PThreads implementation—please refer to section 5.2.3 on page 66 for details.

In an effort to counter this problem I tried to emulate a LIFO queue with the mechanisms of GCD by creating a hierarchy of Dispatch Queues. Please refer to section 4.2.4 for a full description of this technique. Stockfish, however, does not split the search when a new level of the search tree is entered but only after the first move on this level has been completely analysed. This property defeated the original stack emulation scheme, which would have required to suspend the Dispatch Queue associated with the current level of the search tree when the next level is entered. Modifications of the basic search algorithm were not an option either, as discussed in section 4.1.3.

In an alternative approach I adapted the scheme so that the lower-level queues would be suspended whenever a split in the search occurs. This approach does not restore a proper depth-first search, but with only those queues active that are associated with the two deepest levels of the current search tree, the negative impact of the shift towards a breadth-first search would (hopefully) be weaker.

Another design feature of Stockfish, however, defeated even this stripped-down version of the original scheme: Under certain conditions, an active search initiates a supplementary search, which will inevitably start at depth zero of a new search tree. Unfortunately, at this point only those two Dispatch Queues that are associated with the deepest active levels of the "main" search tree may process items. For this reason the new search is bound to freeze when it is split for the first time because the Blocks spawned at the split will be dispatched to (suspended) lower-level queues.

Nevertheless, I had already implemented the queue hierarchy so I decided to use this adapted version of Stockfish for my benchmarks as well. There was at least a chance that even without suspending lower-level queues, the mere presence of the queue hierarchy would effect the order in which Blocks are processed and thus improve the overall performance of the search. The benchmark results (presented in section 5.2.3.3 on page 75), however, did not confirm this assumption.

4.3.3. Optimization is Bad

Stockfish does not only feature algorithmic optimizations. Its developers strive hard to make the best possible use of the traditional threads concept and this includes a well-considered placement of critical variables, many of which are hash tables caching results from processing-intensive evaluations for later reuse. Although these optimizations did not interfere with the remodeling process, their absence resulted in a very poor overall performance of the final GCD version of Stockfish.

Global Data Several essential parameters of the overall search process are held by global variables; most likely for both performance reasons and ease of access. Thread synchronization mechanisms (such as mutexes) as well as the program design itself protect the consistency of this data. An example for the latter approach are variables that are read by all threads but only written by one specific thread. The x86 architecture guarantees that reads and writes of the used data types are carried out in a single operation [Cor13, Section 8.2], which prevents data corruption and makes this scheme safe to use.

The global variables related to the search process prevented the introduction of concurrent searches in the course of the remodeling. A single search, on the other hand, can fully utilize all available resources easily thus further parallelization would probably not benefit the overall performance.

Thread-Local Data The heavy use of caching hash tables turned out to be a much bigger problem than the global data because for performance reasons three of these tables are stored as thread-local variables. The various instances of these tables (one for each thread) are never synchronized with each other. Instead the isolation properties of thread-locality even allow each thread to safely maintain and use pointers to the cached data over the course of the search process without any risk of concurrent modifications.

During my remodeling work on Stockfish, I initially employed what could be called "block-local" versions of these caches: Each newly spawned Block creates its own private version so as to maintain the isolation Stockfish relies on. Fortunately, the underlying hash tables do not require any actual initialization because the values requested by the search process that are not yet available from the cache, will be generated on the fly, stored in the cache and returned just as if the data had been in the cache all along. However, the benchmarks of the resulting implementation exhibited *massive* performance losses (see section 5.2.3.3 on page 75). After further investigation I realized that my approach of creating a new cache instance with every Block lead to a situation where almost none of the values were reused anymore. Soon after or even before the time of

reuse, a split occurs and a new batch of Blocks with empty instances of all formerly thread-local caches is created. The nature of the iterative deepening search algorithm (see also section 4.1.3) with its regular re-evaluation of previously analysed game situations only aggravates the problem.

Restructuring the search to access global versions of the affected caches was out of the question as this would have required adapting large parts of the code. Instead, I employed a trick to reintroduce thread-local storage to Stockfish: Even though GCD does not offer any way to create thread-local storage, the underlying PThread-Workqueue library builds on ordinary PThreads, which do support thread-local storage (see also sections 2.2.2.1, 2.4.3.2 and 3.5). In general it is not predictable which Block will be run by which worker thread. However, in this particular case that does not pose a problem because it does not matter at all which particular cache is used by a specific Block as in this case the only point of thread-locality is to avoid synchronization.

This reintroduction of thread-local, long-living caches did indeed improve performance tremendously. Please refer to section 5.2.3.2 for a detailed discussion of the respective benchmark results.

5. Evaluation

In this chapter I evaluate GCD and my L4Re port of its infrastructure in terms of code size, performance and usability.

5.1. Code Size

In the field of microkernels the size of an application's *trusted computing base* (TCB) plays an important role. The TCB of any application comprises all parts of the system (both hardware and software) that this particular application needs to trust or rely on to faithfully provide its own services. As software is inherently error-prone, a small TCB is preferable for the sake of security and stability.

High-level libraries such as GCD have a significant impact on the TCB; even more so when the library itself is built on lower-level libraries, which is the case for GCD. Therefore, I measured the code size of all libraries required for using GCD on L4Re—please refer to table 5.1 on the next page for the results. While the supporting libraries, Blocks, KQueue and PThread-Workqueue, are rather small, GCD itself adds almost 13,000 SLOC (*Source Lines of Code*) to the TCB. However, not all of the library's code is actually in use when only GCD's core features (i.e., Dispatch Queues and the associated dispatch mechanisms) are used by an application.

Please also note that I deliberately omitted the PThreads library even though it is required to use GCD. Most applications use PThreads to manage threads on L4Re so the library is usually part of a multi-threaded application's TCB anyway and not specific to GCD as the other libraries are.

Aside from the code size of each library, table 5.1 shows the amount of changes I had to make to the respective GNU/Linux version. These are a rough indicator of how expensive (in terms of programming effort) future updates will be—more changes mean more code to merge. I used David A. Wheeler's "SLOCCount" for determining the source code size, whereas the information about changes to the code was taken from the Git revision control system. The values obtained from Git are, in fact, considerable overestimations of the changed code as the mechanism used cannot distinguish between actual modifications to the source code on the one hand and non-functional changes (e.g., mere rearrangement of code or added comments) on the other hand. Just as expected, I managed to keep the changes to GCD to a bare minimum whereas KQueue and especially PThread-Workqueue required a thorough revision to work with L4Re. The vast amount of changes to the PThread-Workqueue library originates from the supplemental code I added to handle system load assessment and load distribution among all available CPUs (see also sections 3.5.2 and 3.5.3).

Library Name (L4Re package)	on L4Re	added	removed
Block Runtime (*libblocks*)	600	0	0
KQueue (*libkqueue*)	2,200	600	400
PThread-Workqueue (*libpthread_workqueue*)	1,700	2,200	1,500
Grand Central Dispatch (*libdispatch*)	12,800	100	20
total	17,300	2,900	1,920

Table 5.1. *The size of each library's implementation on L4Re (in SLOC) and changes in comparison to the GNU/Linux versions (measured in raw lines of text). All values are rounded.*

5.2. Performance

As a new paradigm to express task-level parallelism GCD has to prove its competitiveness against other mechanisms already established in this field; see also section 2.2.2 wherein I introduced the most common techniques. For this purpose, I conducted various benchmark tests that use the two programs described in chapter 4—Fraktal and Stockfish. Both applications originally employed PThreads for task-level concurrency and I created modified versions that built on GCD instead. Measuring and comparing the runtimes of both versions provides a good indicator for evaluating the performance of GCD's infrastructure.

Expectations The original PThreads versions of Fraktal and especially Stockfish are highly optimized for parallel performance with little to no dormant concurrency. Considering the additional complexity of GCD and my approach to make minimal modifications to all involved libraries (see section 3.1), I expected to see a performance penalty for using GCD in comparison to the original PThreads implementation. Most probably this penalty would be significantly higher for Stockfish than for Fraktal, as the former application's remodeling involved blocking operations as part of its recursive splitting of the search (see also section 4.3).

5.2.1. Setup

I conducted my benchmarks on a machine with an Intel Core i5-3550 processor (4 physical/logical cores; 3.30 GHz; 6 MB L3 cache) and about 3 GB of available RAM because 32 bit builds of Fiasco.OC, L4Re and Debian GNU/Linux 7.0 (testing) were used.

Initially I had no intention to use GNU/Linux for my tests, yet massive problems with the GCD version of Stockfish on L4Re forced me to reconsider this approach. Even though Stockfish would start up and run normally on L4Re for up to several minutes, the application reproducibly stopped dead at an arbitrary point during its execution. The effect itself occurred repeatedly, the exact point at which the program would freeze, however, was different during each run and did not follow any recognizable pattern either.

At first, I suspected errors introduced during the remodeling process of Stockfish as there were no such problems with the less complex Fraktal. In an effort to rule this possibility out I ran the GCD version of Stockfish on GNU/Linux, where, to my surprise, it worked without any problems. In combination with the strange timing behavior of the bug, this discovery prompted me to investigate issues related to multi-threading in my L4Re port of PThread-Workqueue because I had barely modified GCD itself and the error seemed to be unrelated to event handling (and therefore to KQueue).

However, I did not find any coding errors and thus kept on using GNU/Linux for further benchmarks of my experimental optimizations for Stockfish to make the best of the situation. Only much later and through extensive testing I found out that the hang-ups were caused by PThread-Workqueue worker threads being assigned higher-than-default scheduling priorities; a constellation that presumably conflicted with L4Re's PThreads implementation; see also section 3.5.1. Although fixing this problem finally allowed me to run the remodeled version of Stockfish on L4Re, the runtimes on L4Re turned out to be extraordinary long (refer to section 5.2.3.2 on page 70) which made a complete rerun of all benchmark tests impossible due to the time constraints on my work. Instead, I discuss some of my experimental optimizations of Stockfish based on the benchmark results gathered on GNU/Linux in section 5.2.3.3 on page 75.

5.2.2. Benchmarks

In sections 4.1.2 and 4.1.3 I introduced Fraktal and Stockfish and briefly described their respective work load. Both programs feature an integrated benchmark mode, which I built upon to collect reliable measurements for my analysis of GCD's performance:

- Fraktal's benchmark computes the Mandelbrot set for a specific, predefined domain of definition. As this calculation takes too little time on modern hardware to allow for reliable results, one benchmark run consists of 40 repetitions for which the total runtime is measured. Once such a cycle is complete the number of worker threads is adjusted and the next cycle starts.
- The benchmark that comes with Stockfish uses a set of 16 predefined game positions and measures the total runtime required for their analysis. Various parameters, such as the stop criterion for the search (mate, maximum search depth, time limit ...) and the number of worker threads to use can be specified before the benchmark starts.

I did not use either of these benchmark directly, but introduced extra wrapper code that sets all relevant parameters (discussed along with the actual measurements in sections 5.2.3.1 and 5.2.3.2 on the following page and on page 70), runs the original

benchmark and takes its runtime. For each individual benchmark configuration ten consecutive runs are measured—I will refer to this pack of ten as *benchmark pass*. By considering the average runtime of a complete pass rather than that of a single run random effects on the measurements can be eliminated (as far as possible). For each pass a single parameter of the respective benchmark is modified. Then one (Stockfish) or three (Fraktal) "warm up" run(s) follow that allow(s) the system, and in particular PThread-Workqueue, to adapt itself to the new work load. The reduced number of warm up runs for Stockfish is due to the much longer runtime for this benchmark.

Because L4Re's scheduler does not automatically distribute threads among the available CPUs (see section 3.5.3), I also had to add supplementary code for this purpose to the original PThreads versions of both programs. Otherwise my evaluation would have compared a GCD version (which is utilizing all the system's CPUs) to a PThreads version that runs on only one CPU. Although Fraktal was designed for L4Re in the first place and thus already included suitable code, it did not allow creating more worker threads than CPUs available in the system. However, one of my benchmark scenarios is designed to intentionally overload the system with threads to emulate real-life situations in which a multi-threaded application expects more CPUs than the system actually has. I replaced the existent CPU-assignment code in Fraktal with a basic scheme for the assignment of workers to CPUs in a round-robin fashion and did the same for Stockfish.

In contrast to PThread-Workqueue (see also section 3.5.3.2) worker termination is not an issue here: While Fraktal never terminates workers at all (idle workers remain blocked in an IPC operation), Stockfish always terminates excess worker threads deterministically in a "created last, terminated first" fashion. In my round-robin scheme I compensated for the latter by adjusting to which CPU a new thread is assigned after a worker has been terminated.

5.2.3. Measurements

Whenever not explicitly stated otherwise, all benchmarks discussed in the following sections were run on Fiasco.OC/L4Re and follow the previously described procedure. Each benchmark pass comprises one or three warm-up run(s) followed by ten consecutive runs whose average runtime constitutes the final result of the pass. These raw results for each benchmark pass are available in appendix A on page vii. Please note, though, that I did not include the more detailed values for each individual run.

5.2.3.1. Fraktal

In the evaluation of Fraktal two benchmark parameters are being used: One parameter both version of the application have in common is the number slices into which the Mandelbrot set's domain is split to distribute the computational load among all workers. Therefore, this partitioning determines the amount of concurrency exposed to the two competing concurrency libraries. For the PThreads version of Fraktal I additionally modified the number of worker threads used in each pass. However, GCD does not offer an equivalent adjustment option because GCD's Concurrent Queues do not limit the number of items running in parallel (see also section 2.4.1). Therefore, I resorted to

a private, undocumented interface of the library (`dispatch_queue_set_width`) that does allow this type of restriction. This way I was able to mirror the original benchmark very closely.

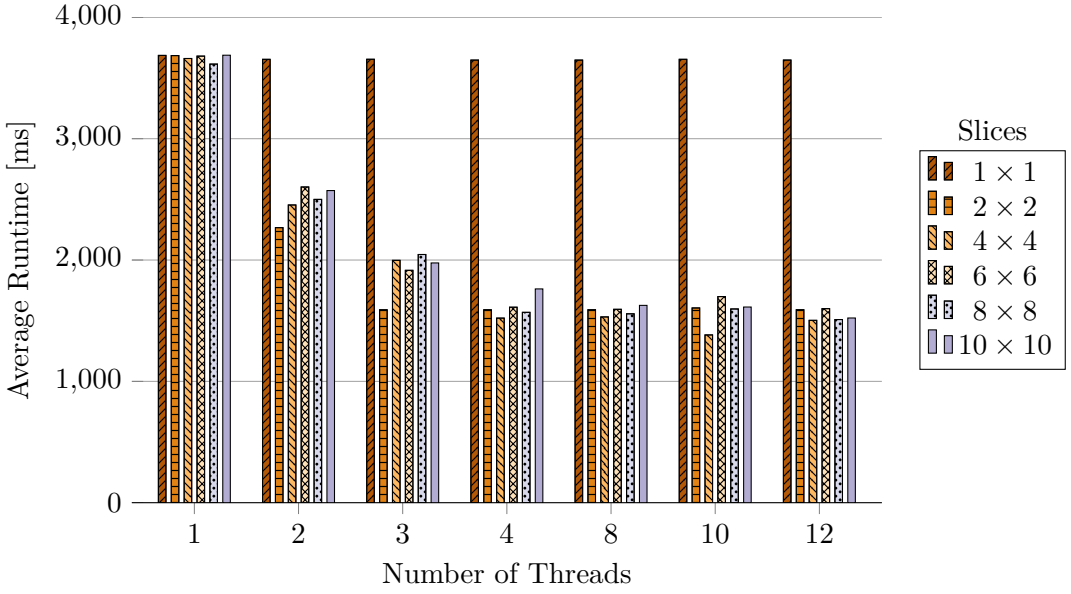
Results Both versions of Fraktal exhibit similar results as illustrated by figure 5.1 on the next page. For the benchmark passes with only one slice (i.e., an undivided domain; indicated in the diagrams by "1×1") the average runtimes are identical. They always amount to about 3.5 seconds, no matter which version of the application is used. The number of employed Blocks/threads is not of importance either as the workload consists of only one item so this benchmark pass equals a sequential version of Fraktal.

When additional threads become available to the PThreads version, the required runtime quickly drops until the minimum of circa 1.5 seconds is reached. At this point there is one worker running on each of the system's four CPUs. Interestingly enough, the total runtime does not increase again, even when there are more active threads than physical CPUs. This behavior is due to the design of Fraktal. The workers do not share any data (among each other) but each worker only interacts with the central managing thread of the application. Therefore the excess workers do not cause any extra synchronization overhead and the scheduling overhead for multiplexing all workers on the available CPUs is negligible in this context.

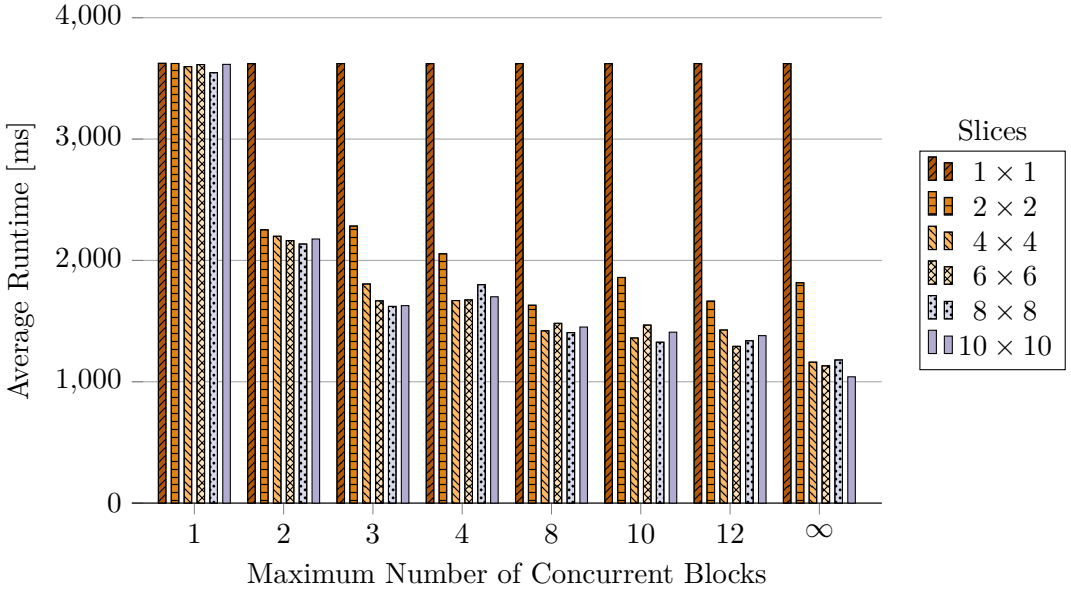
Three of the PThreads benchmark passes are of particular interest: four slices (2×2) with two worker threads, four slices (2×2) with three worker threads and 16 slices (4×4) with 10 worker threads. None of these three measurements fits the overall pattern because their runtimes are better than expected. For instance, there is absolutely no difference between the four-slice passes with either three or four threads even though the latter was able to utilize an additional CPU. My initial assumption was that these peculiarities arise from random effects during the benchmarks, which is why I consulted the full measurement logs that contain the runtimes of each individual run and not just the accumulated average. However, these detailed results, which are shown in figure 5.2a on page 69, do not indicate such an error. On the contrary, there is almost no variance among the different runs of each pass at all. Therefore, I can only suspect the PThreads implementation as the cause of these anomalies.

The measurements of the GCD version are depicted in figure 5.1b and look very similar to those of the PThreads version. With just above 1 second the minimal runtime is even better than that of the PThreads version. Unsurprisingly, this optimum performance is achieved with the maximum number of 100 slices and only with the default configuration of GCD; that is with no arbitrary restriction on the number of concurrently executing Blocks.

Most interestingly, the GCD version of Fraktal shows several abnormal results with four slices (2×2) as well. In contrast to the results for the PThreads version, the runtimes are worse than expected for almost all of these scenarios; two and possibly eight concurrent Blocks being the exception. Again, the individual results for each run (see figure 5.2b on page 69) do not indicate random single errors, which makes the PThreads implementation the most likely source of these anomalies.

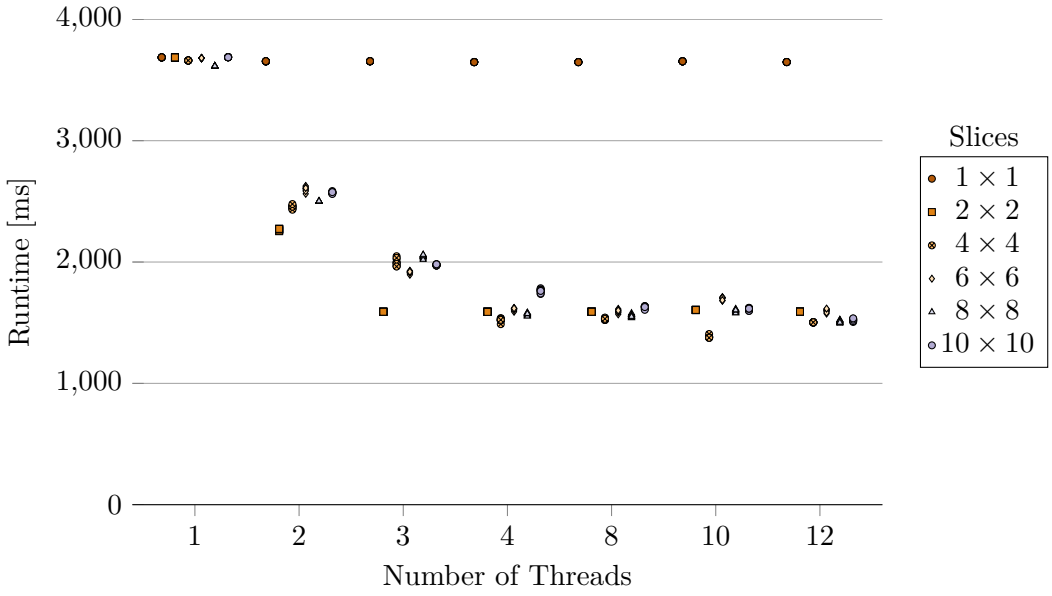


(a) PThreads Version

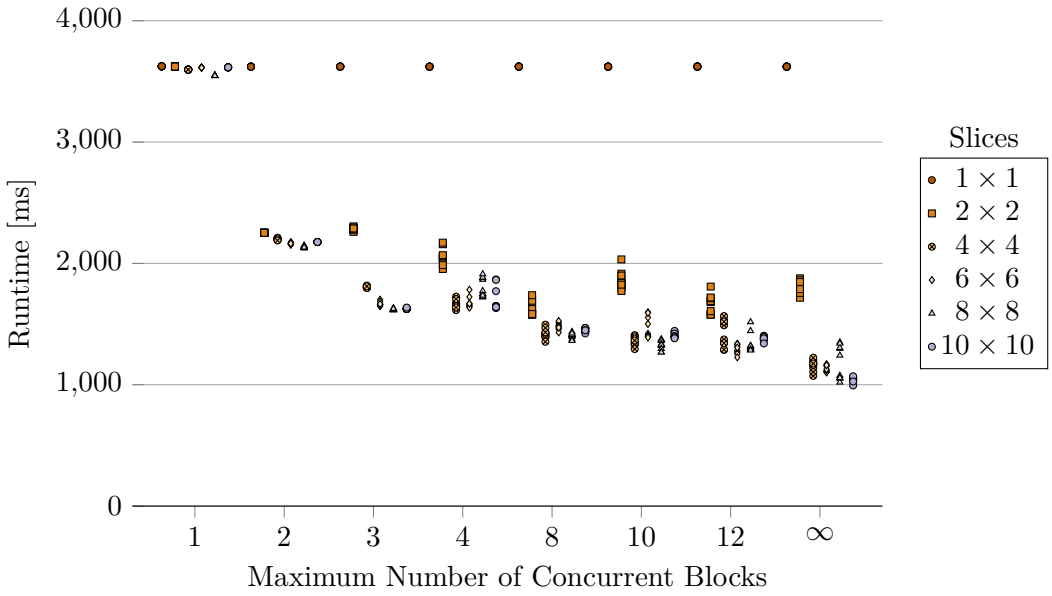


(b) GCD Version

Figure 5.1. *Benchmark—Fraktal [L4Re]*



(a) PThreads Version



(b) GCD Version

Figure 5.2. *Benchmark—Fraktal, Individual Runs [L4Re]*

Nevertheless, the individual runtimes seen in figure 5.2b give an impression of the highly dynamic and unpredictable overall timing behavior of GCD. While the individual runtimes for the PThreads version are almost identical (compare figure 5.2a) they vary significantly with GCD. This effect is even more pronounced with higher limits for the number of concurrent Blocks. Taking the benchmark pass with 64 slices (8×8) and GCD's default configuration as an example the minimum runtime is approximately 1 second, whereas the maximum is more than 1.3 seconds which equals a variance of almost 30%. The reasons for this peculiar behavior can be found in the nature of GCD and its supporting libraries: Blocks that are processed by GCD are handled by three libraries in total—GCD passes them on to PThread-Workqueue, which uses PThreads to finally run the Block's body. This hierarchy presumably allows for more apparently random effects than the PThreads library alone. In addition, PThread-Workqueue manages its worker threads dynamically and without any means to predict future performance parameters, the library will react to changes rather than to expect and prepare for them. Items are assigned to workers on a random basis (see section 3.5.1). Therefore it is possible that in one benchmark run the slices are processed by all existent workers (preventing those workers from becoming idle and being terminated), while in another run some workers remain idle. The manager will then terminate excess idle worker because it has no way of knowing that they will be needed again only moments later. PThread-Workqueue already implements mechanisms to avoid such thrashing (see section 3.5.2.3), but it cannot be prevented completely.

5.2.3.2. Stockfish

In the Stockfish benchmark tests only a single parameter was modified over the course of all passes. For the original PThreads-based version this parameter was the (maximum) number of worker threads available for the search. The parameter for the GCD version of Stockfish is a byproduct of the application's remodeling: the *split factor*, that is the number of Blocks that are spawned each time the search is split (see section 4.1.3). A split factor of one indicates that no actual split occurs and the application effectively runs single-threaded. Please note, that I used an optimized GCD version of Stockfish which incorporates thread-local storage (see section 4.3.3) for all benchmarks discussed in this section. In section 5.2.3.3 on page 75 I will also present measurements of the initial, unoptimized version for comparison.

My benchmarks of Stockfish started out on GNU/Linux because the GCD version reproducibly froze on L4Re (see section 5.2.1) and it was not possible to conduct even a single full execution of all benchmark series. Although I was able to run the application on L4Re later on, I present the data collected on GNU/Linux as well because I deem these a better representation of the remodeled version's actual performance. I will explain my reasons for this assessment in the following two paragraphs when discussing the respective results.

Results on GNU/Linux The measurement results for the original PThreads implementation (figure 5.3a on page 72) show the expected V-shape: Starting at 4 seconds with a single thread, additional workers gradually improve the performance until the optimum

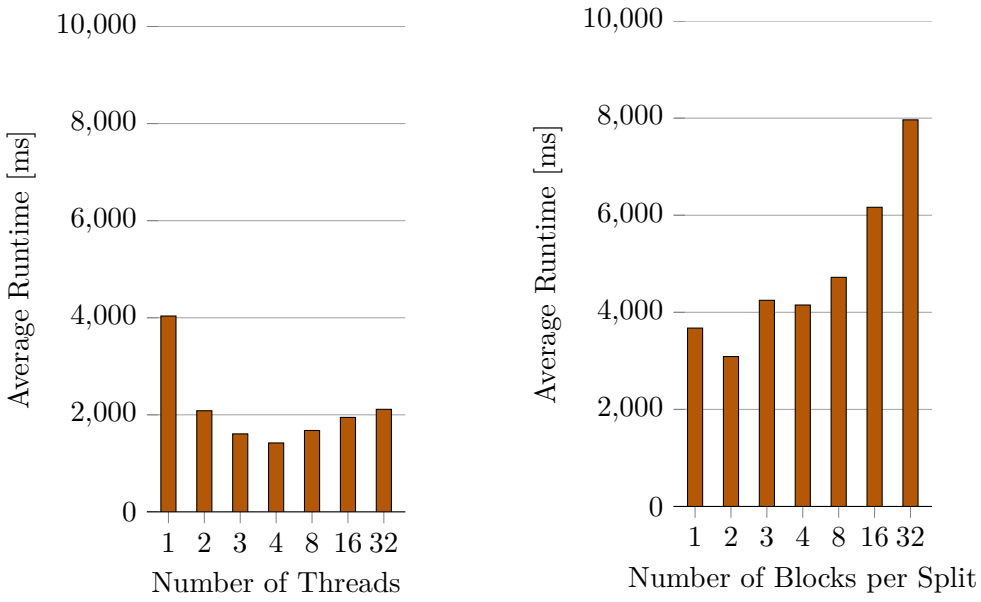
of about 1.4 seconds is reached with four threads. At this point, one thread runs on each physical CPU core of the system. When more threads are added, the runtime increases again, up to a maximum of approximately 2 seconds; the same value that is achieved with two threads. Apparently, with more active workers than CPUs the additional synchronization dominates as all CPUs are already fully utilized. Remember that *all* threads involved in a given split communicate among themselves (quite frequently) via a single shared data structure, which is protected by a PThreads mutex. In fact, even those benchmark passes with less than four threads show this overhead as the performance improvement for additional threads gets smaller and smaller. While the transition from a single thread (4 seconds) to two threads (2 seconds) halves the runtime, the addition of a third (1.6 seconds) and fourth thread (1.4 seconds) effect ever smaller improvements.

The values for the GCD version (figure 5.3b on the following page) start out with a surprise: Running single threaded (i.e., with a split factor of one), the remodeled application has a runtime of only about 3.7 seconds and is therefore actually faster than the PThreads version. This effect originates from the more efficient synchronization mechanisms of the GCD library: Text output to the console, for example, uses the I/O-Streams of the C++-STL which are not thread-safe. Therefore, Stockfish applies safe-guards against a concurrent usage in the form of a mutex (PThreads) or a dedicated Serial Dispatch Queue (GCD). Even with just a single active thread or Block this protection is in place and affects performance.

Further results are less positive, though. The performance of the remodeled application is significantly worse than that of its original implementation. Although the runtime is reduced to approximately 3.1 seconds with a split factor of two, this is already the optimum value yet far from the 1.4 seconds the PThreads version can offer. Apparently, the remodeling process disabled essential performance optimizations of Stockfish, some of which I discussed in section 4.3.3.

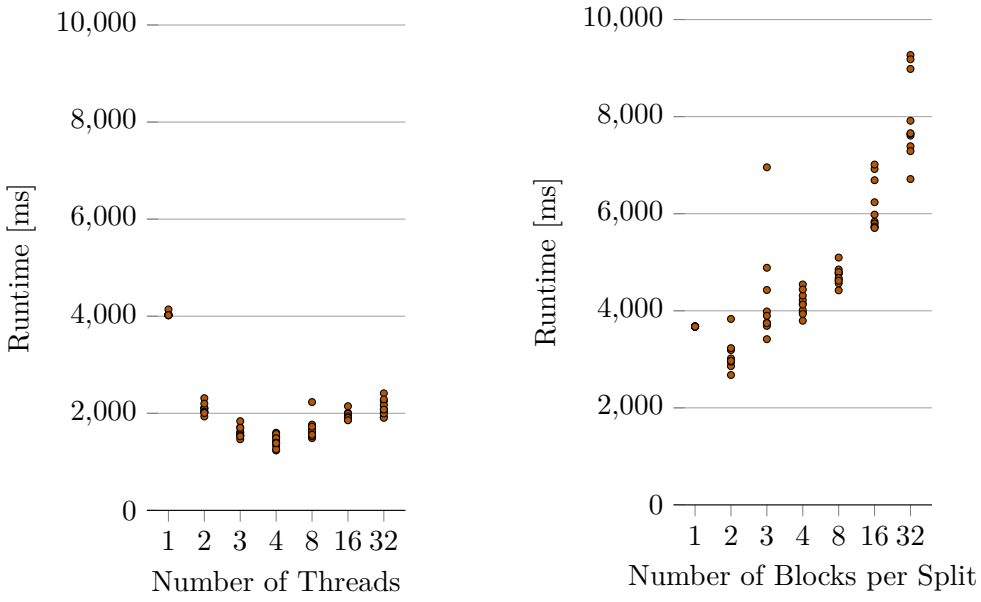
With higher split factors the runtime deteriorates even more. The reason for this behavior is the exponentially growing number of Blocks in existence because each Block may be split again during the search process. Due to the FIFO property of Dispatch Queues, larger numbers of Blocks worsen the shift from a depth-first search towards a breadth-first search and thus diminish the accelerating effects of search optimizations such as alpha–beta pruning; see also section 4.3.2. At the same time, a large number of active Blocks results in the blocking of many worker threads. This is due to the suboptimal design of Stockfish's GCD version, which potentially blocks a worker each time the search is split (see section 4.1.3). Even though PThread-Workqueue compensates for blocked workers by spawning new ones, this reaction takes time and incurs management overhead. What is worse, a new worker thread starts out with its thread-local caches empty (see section 4.3.3) and thus is likely to perform much worse than the now-blocked worker it replaces. Please refer also to section 5.2.3.3 on page 75 in which I explain the effects of these caches in detail.

A split factor of three causes a disproportional runtime increase. To investigate this phenomenon further, I once again consulted the full benchmark protocol mentioned in the previous paragraph and analyzed the individual results of all benchmark runs. These values are shown in figure 5.4b on the following page and draw quite a different picture.



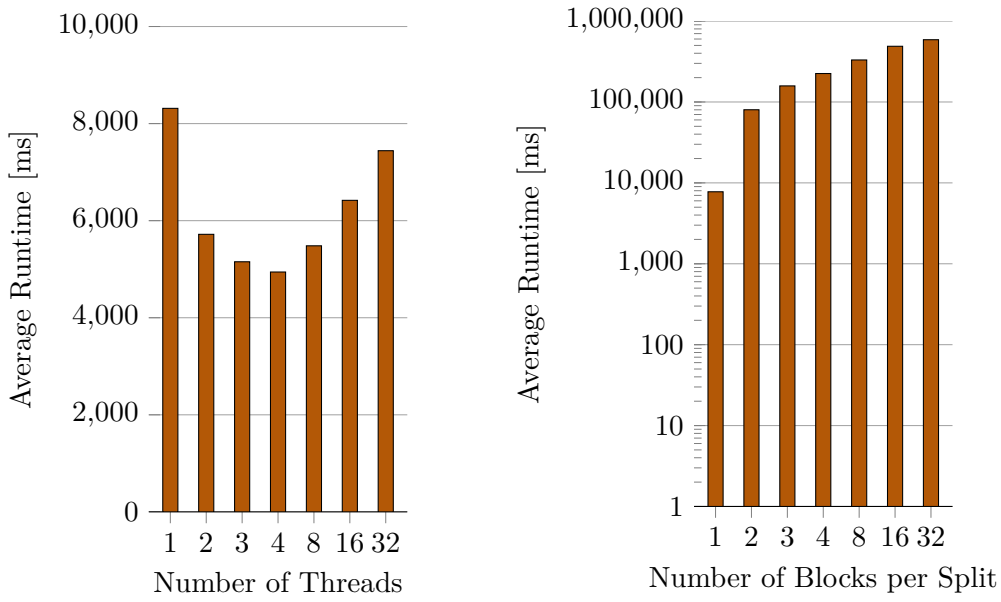
(a) PThreads Version *(b) GCD Version*

Figure 5.3. *Benchmark—Stockfish [GNU/Linux]*



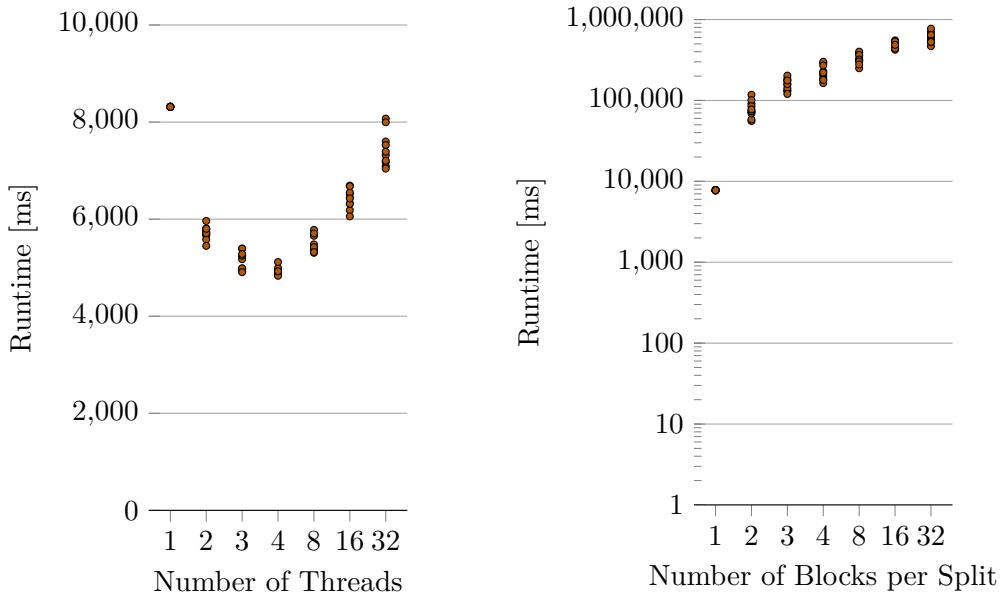
(a) PThreads Version *(b) GCD Version*

Figure 5.4. *Benchmark—Stockfish, Individual Runs [GNU/Linux]*



(a) *PThreads Version* ***(b)*** *GCD Version*

Figure 5.5. *Benchmark—Stockfish [L4Re]*



(a) *PThreads Version* ***(b)*** *GCD Version*

Figure 5.6. *Benchmark—Stockfish, Individual Runs [L4Re]*

Ignoring the spikes for a moment, the early passes show overall lower runtimes and the results with a split factor of three (which, for reasons unknown, exhibits particularly many extremely high values) fits the general trend of increasing results.

In addition, figure 5.4b is another good example for the unpredictability of GCD—for the pass with a split factor of three the minimum (3.4 seconds) and the maximum (7 seconds) differ by slightly more than 100%. I already discussed this effect and its origins at the end of section 5.2.3.1 but seeing the same behavior on GNU/Linux backs my assumption that this is an inherent trait of GCD or at least of the user space port of its supporting infrastructure. Please note that the PThreads-based version of Stockfish exhibits only insignificant runtime variations (see figure 5.4a on page 72) because in this case all of the application's optimizations work as intended and workers threads are fully under the control of the application.

Results on L4Re All runtimes measured on L4Re (depicted in figure 5.5 on the preceding page) are significantly longer than those on GNU/Linux, both for the PThreads version and the GCD version. The former's runtime with a single thread is about 8.3 seconds; the optimum value with four concurrent threads is around 5 seconds. Very similar to the results on GNU/Linux adding a second thread has the greatest influence on performance—it reduces the runtime from 8.3 seconds to 5.7 seconds—whereas the third and fourth thread bring only minor improvements (5.2 seconds and 4.9 seconds respectively). When introducing more threads than physical CPUs the resulting performance degradation is more pronounced this time with the 32-thread pass (7.4 seconds) being almost at the same level as the single-threaded pass. These results imply that the PThreads synchronization overhead has a much greater impact on L4Re than on GNU/Linux. The more detailed results (see figure 5.6a on the previous page) demonstrate that the variance of individual runs, too, is higher on L4Re because this time even the PThreads version of Stockfish shows considerable variance.

When the remodeled version of Stockfish finally worked properly on L4Re its measurements were disappointing. In contrast to the Fraktal benchmarks, at which the GCD version fully matched the original PThreads implementation, the runtimes of Stockfish's GCD version are extraordinarily high. Please note that figure 5.5b on the preceding page, which shows the measurements for the GCD version, uses a logarithmic ordinate! Interestingly enough, with a split factor of one (i.e., running single threaded) the GCD implementation (7.8 seconds) is faster than the original PThreads version (8.3 seconds); the same effect occurred on GNU/Linux. At the moment when the search is actual split the runtime skyrockets to 80 seconds for a full benchmark pass with a split factor of two, continues to increase nearly exponentially and hits 590 seconds (close to 10 minutes!) with 32 Blocks created per split.

Even though the tests with the PThreads version of Stockfish have already established the inferior performance of L4Re's PThreads implementation, this alone cannot explain these devastating results. A much larger contribution originates from my PThread-Workqueue port which lacks an optimization for the fast replacement of blocked worker threads as the GNU/Linux port of the library features; see section 3.5.2.3. Due to its suboptimal remodeling, in which I was forced to use blocking operations as an integral

part of the search splitting process (see also section 4.2.1), Stockfish constitutes a perfect pathological example: Once all available workers are blocked it can take up to one second before the PThread-Workqueue manager spawns an additional worker (compare section 3.5.2.3). Unfortunately, it is very likely that this new worker will immediately execute a Block that subsequently tries to split the search again and gets blocked in the process. Since the manager will never spawn more than one worker per second it can take quite some time for the situation to normalize. During the benchmarks these delays were actually visible: all output to the screen stopped for several seconds until a sufficient number of new workers were available and Stockfish resumed its normal operation.

In section 5.2.3.3 I present the results of an additional benchmark which I had designed to confirm that the described situation is indeed the cause of the major performance loss observed with Stockfish's GCD version on L4Re.

5.2.3.3. Optimizations

During the evaluation of GCD's performance I also tested various optimizations, both for the PThread-Workqueue library on L4Re and on Stockfish, as this application showed a particularly bad performance. Due to the problems with running the latter on L4Re (discussed in section 5.2.1) I used Fraktal for my optimization experiments with PThread-Workqueue while all optimizations of Stockfish's GCD version had to be tested on GNU/Linux. Only during the final stages of my work, when Stockfish finally worked properly on L4Re, I conducted an additional benchmark with the application to back my assumptions (discussed at the end of section 5.2.3.2) on the origin of its bad performance on this system.

Worker Thread Migration Normally, the worker threads of the PThread-Workqueue library are assigned to a CPU when they are created and this relationship never changes during the worker's whole lifetime. In section 3.5.3.3 I presented an approach to a more dynamic assignment that would likely provide a better load balancing among all CPUs. However, running the Fraktal benchmark with active worker migration (see figure 5.7 on the following page) did not show any significant change in the measured runtime compared to the results without migration (see figure 5.1b on page 68). The anomalies discussed in section 5.2.3.1 are more pronounced when workers migrate between CPUs, which may indicate that they originate from within the system's scheduler. More experiments would be necessary to reliably pin-point the source though.

Dispatch Queue Hierarchy After it became clear that the approach of combining a Dispatch Queue hierarchy with queue suspension cannot be used to reestablish the proper depth-first search of Stockfish (see sections 4.2.4 and 4.3.2), I decided to make the best of the invested work and tried to use the queue hierarchy without suspending lower-level queues. The official GCD documentation [Incb] does not specify which effect such a hierarchy has, though. My assumption was that even without suspension the hierarchy would impose a priority on Blocks according to the level they are dispatched to. Provided such priorities exist, those Blocks which enter the hierarchy at the level

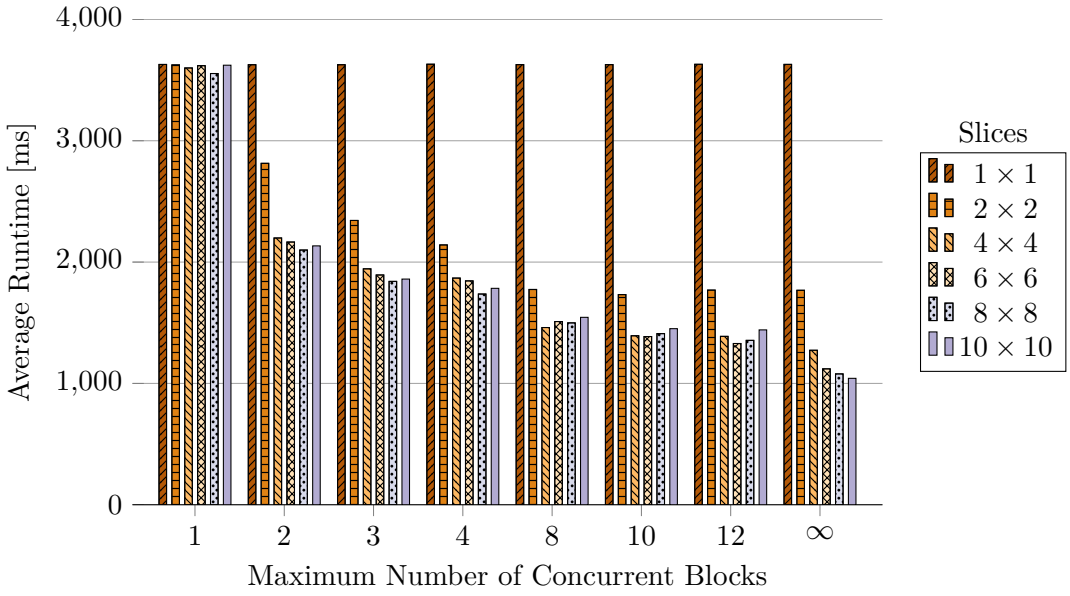
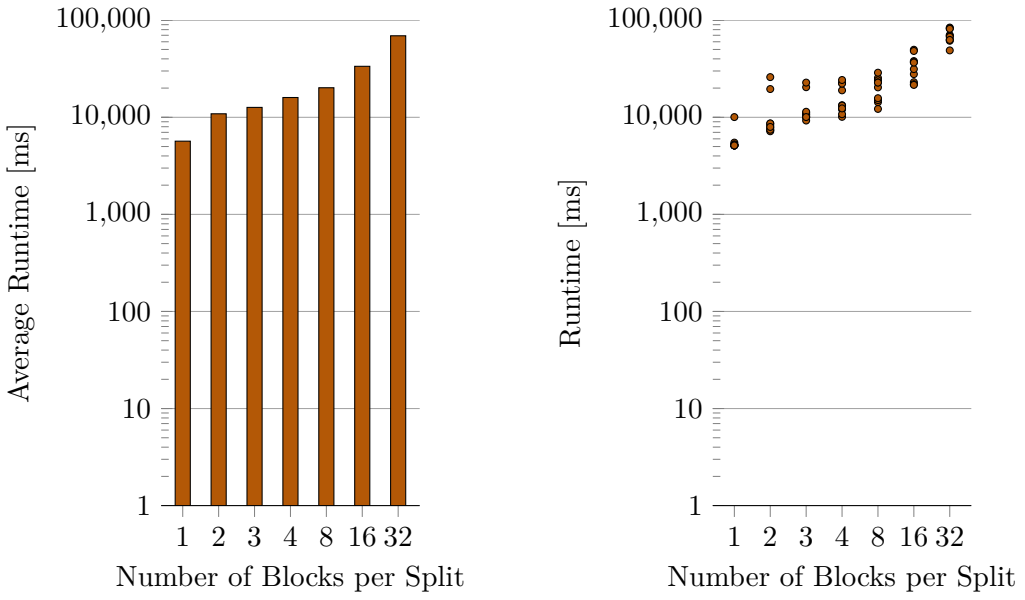
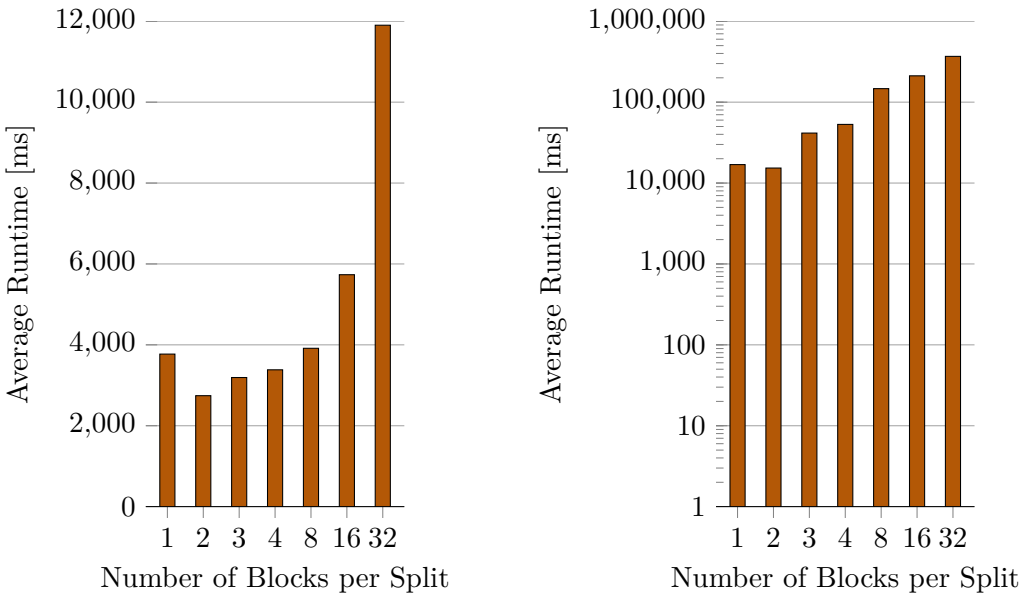


Figure 5.7. *Benchmark—Fraktal, GCD Version, Worker Migration [L4Re]*



(a) *Passes* ***(b)*** *Individual Runs*

Figure 5.8. *Benchmark—Stockfish, GCD Version, 200 Standby Workers [L4Re]*



(a) *With Dispatch Queue Hierarchy* ***(b)*** *Without Thread-Local Storage*

Figure 5.9. *Benchmark—Stockfish, Optimizations of GCD Version [GNU/Linux]*

"closest to the final queue" would most likely be processed first. To check the validity of this assumption and possibly assess the overall effect of such a queue hierarchy, I conducted a benchmark with a version of Stockfish which I had adapted according to the scheme fully described in section 4.3.2.

As can be seen by comparing figure 5.3b on page 72 and figure 5.9a, there is no clear result. Whereas the queue hierarchy is apparently beneficial for small split factors, it increases the variance seen with large split factors and even increases the runtime of the benchmark pass with 32 Blocks per split (12 seconds instead 8 seconds). It is as though the hierarchy does indeed have an influence on the order in which Blocks are processed by GCD but also imposes overhead on its own right, which results in performance losses when greater numbers of Blocks are used.

Thread-Local Storage As stated at the beginning of section 5.2.3.2, all previously discussed benchmarks of Stockfish incorporate my first optimization—the introduction of thread-local storage for the hash tables that cache evaluation results of game positions; see also section 4.3.3. I decided to keep this optimization in place because the performance of the GCD version of Stockfish was extraordinarily poor without it. Further benchmarks would have been much too time-consuming at these performance levels.

Figure 5.9b shows the benchmark results without thread-local storage to give an impression of the effect. Please note the logarithmic ordinate! A direct comparison with figure 5.3b on page 72 clearly indicates the massive improvement of up to factor 46 as seen for the scenario with 32 Blocks per split. Without thread-local storage this particular pass took almost 370 seconds on average and with the optimization only 8 seconds.

Enlarged Pool of Standby Workers When discussing the poor performance of Stockfish on L4Re near the end of section 5.2.3.2, I had speculated that one major cause for these results was the lack of a special optimization for spawning new workers in the PThread-Workqueue library. In an attempt to substantiate this claim I conducted an additional benchmark test for which I manually modified the library to increase the number of standby workers (see also section 3.5.1) from the current default of two to 200. This adjustment mitigates the delay caused by blocked workers because (in most cases) there are enough workers on standby which can immediately replace the blocked ones.

Figure 5.8 on page 76 illustrates the results of this benchmark, which indicate a massive improvement in direct comparison to the initial measurements (shown in figure 5.5b on page 73), especially when small split factors are used. Apparently the increased number of workers is sufficient in these scenarios whereas for higher split factors even more workers are needed, which causes the familiar delays.

Peculiarly, even the runtime of the single-threaded benchmark pass (split factor one) improved noticeably from 7.8 seconds to 5.7 seconds with the modified version. Theoretically my modification of the PThread-Workqueue library should have no influence on this particular pass at all. On the other hand, Stockfish's main search is not the only activity in the remodeled application that uses GCD. There are also management functions such as the regular checking of the search time (so that a time-limited search can be stopped).

However, this result suggests that the current setup with just two standby workers should probably be adapted by increasing the number of such workers. At the very least the experiment confirms that additional optimization of the PThread-Workqueue port on L4Re is advisable.

5.2.4. Interpretation

First and foremost, the results of the Fraktal benchmarks prove that my L4Re port of GCD is able to deliver a comparable performance to PThreads. According to the tests with Stockfish, however, that situation can change drastically under unfavorable circumstances. As a matter of fact, the benchmarks with Stockfish only confirmed what I had already suspected during the remodeling process—the application's design is incompatible with the basic paradigm of GCD and this inevitably leads to patchwork solutions and to unsatisfactory performance.

The particular case of Stockfish turned out to be the worst possible combination of an application design, whose serial structure forced me to introduce blocking operations into the GCD version and my PThread-Workqueue implementation which is lacking an optimization for this exact situation. In addition, Stockfish also showed how big an impact relatively simple optimization can have. Increasing the number of standby workers that PThread-Workqueue retains as well as reintroducing thread-local storage both improved the overall performance of Stockfish by an order of magnitude.

A second important insight concerns the general nature of GCD itself. The detailed results of all individual benchmarks runs indicate significant runtime variations although exactly the same application was run. Even though similar behavior occurred with the PThreads version of Stockfish, the variations were much more pronounced whenever GCD

was involved. Apparently the library (or more likely the underlying PThread-Workqueue) introduces highly dynamic and unpredictable timing characteristics into the application built on it.

In conclusion, the GCD infrastructure I established on L4Re delivers good performance results for applications that are well suited to the design paradigms of GCD, while it may severely affect those applications that are not.

5.3. Usability

Aside from efficient use of available hardware resources GCD's other main objective is to provide application programmers with a convenient and simple access to concurrency.

I indeed managed to remodel both programs I worked with in such a way that there is no trace of explicit thread usage left and the one major difference between serial and concurrent execution of code is simply the employed Dispatch Queue. Blocks offer an exceptionally concise way to pass functions on to GCD for execution and (automatically) include all necessary data at the same time. Furthermore, the absence of mutexes in the restructured parts of the applications make the code more comprehensible and prevent careless mistakes, such as forgetting an unlock operation. Stockfish, in particular, employed mutexes to protect the consistency of data structures used concurrently by its worker threads. As part of the program's optimizations the corresponding `lock` and `unlock` operations were scattered across multiple screens of source code and even across multiple source files. This resulted in a type of code that is very hard to comprehend and very prone to errors. The replacement of mutexes with Serial Queues eliminated these problems and even improved performance as indicated by the single-threaded benchmark passes.

On the other hand, the previously described benefits of Blocks are, in fact, not bound to GCD or even threads. Any program written in C, C++ or Objective-C may utilize the technology as long as the compiler in question supports it. And while GCD's simplifications and automatisms indeed facilitate the developers' work, they also deprive them of full control over their applications and involve highly unpredictable timing behavior. Especially Dispatch Queues, the core concept of GCD, offer only very limited functionality which can become a problem if there is a need for more advanced semantics as in the case of Stockfish.

In other words: GCD's simplicity is the library's greatest advantage and at the same time its greatest weakness.

6. Conclusion

In this chapter I want to bring this thesis to a close with a short summary of my work and an overview of loose ends that might be good starting points for follow-up projects. Furthermore, I will discuss possible options for the direct integration of *Grand Central Dispatch* (GCD) into the Fiasco.OC microkernel.

6.1. Summary

Over the course of my thesis, I successfully ported GCD to the *L4 runtime environment* (L4Re) by adapting its supporting libraries (chapter 3). In addition, I integrated Block literals into the build infrastructure of L4Re to allow for convenient use of the technology. Thanks to this approach, I managed to avoid making substantial changes to GCD itself (section 5.1), which will facilitate incorporating future updates of this fast-evolving library.

Considering the intended scope of this work I had to omit most Dispatch Sources, as porting them would have required considerable modifications of otherwise unrelated parts of L4Re (section 3.4.2). Even though not the full feature set of GCD is available on L4Re, the core functionality of the library, most notably Dispatch Queues and the related features, is fully operational and provides a solid base for experiments with the technology.

After the porting and integration process was completed, I evaluated the performance of GCD—in general and for my L4Re port in particular—by restructuring two existent applications to use GCD instead of *POSIX Threads* (PThreads) (chapter 4). The results look promising (section 5.2), especially taking into account that my choice of applications was not optimal in retrospect as both were heavily optimized for PThreads. However, one of them turned out to feature an overarching design that is largely incompatible with the paradigm of GCD, which resulted in severe performance losses (section 5.2.3.2).

One particularly remarkable finding were the extremely unpredictable runtimes of the GCD-based versions of both applications. Multiple benchmark runs with exactly the same parameters resulted in vastly different runtimes due to the dynamic nature of the PThread-Workqueue library and the way it manages its worker threads (sections 3.5 and 5.2.3). Such behavior can become a major problem for time-critical or real-time applications, whose scheduling requires a precise estimation of their worst case execution time.

Further optimization is necessary before GCD can be considered a serious match for PThreads on L4Re in terms of performance. The library does, however, offer substantial benefits when it comes to usability and the avoidance of careless mistakes.

6.2. Kernel Integration

There have been proposals to incorporate GCD into the Fiasco microkernel itself; possibly even as a full replacement for traditional threads. As Blocks are all but completely handled by the compiler (compare section 3.2) the question of kernel integration does not arise for them.

From what I have learned over the course of my work, the official GCD library is not suitable for integration into a microkernel: The library's source code is very complex, sparsely documented and subject to frequent changes on account of the ongoing development. These facts were the reason for me to retain KQueue and PThread-Workqueue as independent libraries on L4Re instead of modifying GCD to incorporate their functionality and use the system's interface directly (see section 3.3.2). Consequently, I would advise against merging the library into Fiasco.OC.

A preferable, yet labor-intensive approach is to reimplement and integrate only selected parts of GCD's core functionality—namely Dispatch Queue and the associated dispatching framework—into Fiasco.OC. These features would be sufficient to replace traditional threads as the first class scheduling primitive of the kernel. In contrast to the full-fledged GCD library, a version reduced to its bare essentials in this manner, is also compliant with the minimalistic approach of a microkernel (section 2.1.1). Supplementary features can be provided through a user-space library, which is based on and compatible with the official release of GCD.

Somewhere in between a full kernel integration of GCD and recreating parts of its functionality from scratch within the kernel, there is a third option: incorporating PThread-Workqueue into the system's scheduler. This library is significantly simpler than GCD but directly affects the latter's performance. In addition, changes to the interface of PThread-Workqueue are much more unlikely than changes to GCD. When managed by the scheduler—the central CPU coordination facility of the system—adapting the size of the worker pool to the current system load and available resources becomes both simpler and more reliable than with the current, indirect solution (section 3.5.2). Although PThread-Workqueues are much more basic than the Dispatch Queues of GCD, they too have the potential to completely replace traditional threads and thus preserve the minimalistic feature set of Fiasco.OC.

6.3. Future Work

My L4Re port of GCD is well suited for experimental use, but for practical application it lacks features of the official library release as well as performance optimizations for L4Re. The following ideas focus on these aspects which are the foundations for possible future works. In general, optimizations of PThread-Workqueue will translate to performance improvements for applications built on GCD whereas enhancements of KQueue are required to complete the feature set of the library.

PThread-Workqueue My work was focused on GCD and that is why I concentrated on porting PThread-Workqueue to L4Re because the library is a requirement to run GCD in its unmodified form. The optimization of PThread-Workqueue,

however, was not at the core of my interest. Nevertheless, due to the central role of the library in regard to the performance of GCD I had to apply some basic optimization.

It would be interesting to explore this path further and tweak the way the library assesses system load, adapts the pool of workers and distributes them among the CPUs of the system (sections 3.5.2 and 3.5.3). Aside from the previously mentioned merge of the library with the system's scheduler, machine learning is a promising approach as well. Observing an application's usage patterns might permit the library to infer future computational demands of said application. Using this information, the worker pool manager is able to improve the runtime adaptation of threads; for example it can raise the number of standyby workers if new items are expected shortly.

KQueue My current L4Re port of the KQueue library features only those Filters required for GCD (see section 3.4.2). Nevertheless, additional Filters would be convenient—especially those that are subsequently accessible through one of GCD's Dispatch Sources.

Even Filters/Dispatch Sources that do not seem useful on L4Re can be of interest, because they could be given new semantics. `EVFILT_MACHPORT` for example, which originally monitors Mach ports[1] may be adapted to work with L4Re's IPC-Gates instead.

Acquisition of Information Serial Dispatch Queues inherently express dependencies among the items dispatched to them: When item A follows item B in a Serial Queue, then B is (indirectly) dependent on A because of the queue's properties (see also section 2.4.1). This implicit information can be retrieved and used for scheduling decision, automatic deadlock detection or the like.

Concurrent Queues, on the other hand, hold information about the inherent parallelism of an application—both in terms of quantity ("How many items are in the Queue?") and temporal variance ("How does this number change during the runtime of the application?"). By instrumenting the queues, this information can be extracted, and analyzed to facilitate further optimization.

Block literals, too, have latent potential in this regard. A modified compiler may be able to extract information about the runtime behavior such as the Block's expected (best case/average case/worst case) execution time or the data usage patterns of the Block's body. The operating system can then make use of this extra information at runtime to adjust its scheduling decisions or assign a Block to a special-purpose processor (such as a GPU). As an alternative to the automated extraction by the compiler, the required metadata could also be provided by the developer through annotations on the Block.

[1] Mach ports are endpoints for the IPC mechanism of the Mach microkernel. XNU, the kernel of Apple's Mac OS X operating system, is partly based on Mach and incorporates Mach IPC as well.

Enhanced Dispatch Queues Currently there are two types of Dispatch Queues available in GCD, which differ only in the number of items that are executed in parallel (see section 2.4.1). This concept can be extended to offer a greater variety of queues—for example queues that run all their items on a special-purpose processor or provide automatic replication as a safeguard against hardware failures.

An upgrade of the existing dispatch mechanism would be of interest as well. More control over the dispatching process, such as the means to insert new queue items at arbitrary positions of the queue or cancel queued items later on, would allow a wider range of application for Dispatch Queues.

Glossary

Block

Block Literal—Non-standard extension to the C programming language family; introduces the concept of anonymous/lambda functions to these programming languages

Clang

Compiler frontend for the C programming language family; uses LLVM as backend and supports Blocks

Dispatch Queue

Central interface of GCD; a work queue that accepts pieces of code (Blocks for example) and, with the help of an automatically managed thread pool, runs them in FIFO order

Fraktal

A small multi-threaded application available on L4Re that calculates the Mandelbrot set and, optionally, displays a graphical representation thereof

Grand Central Dispatch

Alternative paradigm for expressing task-level parallelism; centered around the concept of Dispatch Queues instead of native threads

KQueue

Efficient and highly-scalable event monitoring facility; originally implemented in the FreeBSD kernel

PThread-Workqueue

Non-standard extension to the PThreads library; provides a kernel-managed thread that dynamically adapts its set of worker threads to system load

SLOC

Source Lines of Code—Metric used to measure the code size of software; all figures in this work were generated using David A. Wheeler's "SLOCCount" (`http://www.dwheeler.com/sloccount/`)

Stockfish

An open source chess engine, released under the GPLv3 license

Acronyms

AI artificial intelligence

DSP digital signal processor

FIFO first-in, first-out

FPGA field-programmable gate array

GCC GNU compiler collection

GCD Grand Central Dispatch

GPU graphics processing unit

HPC high-performance computing

L4Re L4 runtime environment

LLVM Low Level Virtual Machine

MPI message passing interface

Open MP Open Multi-Processing

POSIX Portable Operating System Interface

PThreads POSIX Threads

RAII resource acquisition is initialization

SIMD single instruction, multiple data

SSE streaming SIMD extensions

STL Standard Template Library

TBB Intel Threading Building Blocks

TCB trusted computing base

References

[Amd67] Gene Myron Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of AFIPS Spring Joint Computer Conference*. 1967, pp. 483–485. URL: http://www-inst.eecs.berkeley.edu/~n252/paper/Amdahl.pdf (visited on 02/24/2013).

[ARB11] The OpenMP Architecture Review Board (OpenMP ARB). *OpenMP Application Program Interface – Version 3.1*. July 2011. URL: http://www.openmp.org/mp-documents/OpenMP3.1.pdf (visited on 02/23/2013).

[Bro+10] Andre Rigland Brodtkorb et al. "State-of-the-art in heterogeneous computing". In: *Scientific Programming* 18.1 (2010), pp. 1–33. ISSN: 1875-919X. URL: http://info.ornl.gov/sites/publications/Files/Pub24800.pdf (visited on 02/20/2013).

[Bur+97] Jim Burns et al. "A Dynamic Reconfiguration Run-Time System". In: *FPGAs for Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on*. IEEE. 1997, pp. 66–75. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.43.532&rep=rep1&type=pdf (visited on 02/20/2013).

[Cho+01] Andy Chou et al. "An empirical study of operating systems errors". In: *Proceedings of the eighteenth ACM symposium on Operating systems principles*. 2001, pp. 73–88. URL: http://citeseer.ist.psu.edu/viewdoc/download?doi=10.1.1.21.6489&rep=rep1&type=pdf (visited on 01/23/2013).

[Cor] Intel Corporation. *Intel Threading Building Blocks (Intel TBB)*. URL: https://threadingbuildingblocks.org/ (visited on 02/26/2013).

[Cor13] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Vol. 3A: System Programming Guide, Part 1. Mar. 2013. URL: http://download.intel.com/products/processor/manual/253668.pdf (visited on 04/07/2013).

[Dev] Bob Devaney. *The Mandelbrot Set Explorer*. URL: http://math.bu.edu/DYSYS/explorer/def.html (visited on 03/03/2013).

[Drea] OS Group Dresden. *L4Re – The L4 Runtime Environment*. URL: https://os.inf.tu-dresden.de/L4Re/ (visited on 03/02/2013).

[Dreb] OS Group Dresden. *The Fiasco.OC µ-kernel*. URL: https://os.inf.tu-dresden.de/fiasco/ (visited on 03/02/2013).

[Gro08] Austin Common Standards Revision Group. *POSIX.1-2008*. Also published as IEEE Std 1003.1-2008 and ISO/IEC 9945:2009. The Open Group, 2008. URL: http://www.opengroup.org/onlinepubs/9699919799/ (visited on 02/25/2013).

[Hut] Nick Hutchinson. *libdispatch—Linux port of Apple's open-source concurrency library*. URL: https://github.com/nickhutchinson/libdispatch (visited on 11/12/2012).

[Inca] Apple Inc. *Concurrency Programming Guide*. URL: https://developer.apple.com/library/mac/documentation/General/Conceptual/ConcurrencyProgrammingGuide/ConcurrencyProgrammingGuide.pdf (visited on 11/13/2012).

[Incb] Apple Inc. *Grand Central Dispatch (GCD) Reference*. URL: https://developer.apple.com/library/mac/documentation/Performance/Reference/GCD_libdispatch_Ref/GCD_libdispatch_Ref.pdf (visited on 11/13/2012).

[LA06] Jork Löser and Ronald Aigner. *Building Infrastructure for DROPS (BID) Specification*. 2006. URL: https://os.inf.tu-dresden.de/l4env/doc/html/bid-spec/ (visited on 05/07/2013).

[LC] Mark Lefler and the CPW team. *Chess Programming Wiki*. URL: https://chessprogramming.wikispaces.com/ (visited on 02/24/2013).

[LCD91] David Levine, David Callahan, and Jack Dongarra. "A comparative study of automatic vectorizing compilers". In: *Parallel Computing* 17.10 (1991), pp. 1223–1244. URL: ftp://info.mcs.anl.gov/pub/tech_reports/reports/P218.pdf (visited on 02/13/2013).

[Lee06] Edward Ashford Lee. "The Problem with Threads". In: *Computer* 39.5 (2006), pp. 33–42. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf (visited on 02/20/2013).

[Lem01] Jonathan Lemon. "Kqueue: A generic and scalable event notification facility". In: *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*. 2001, pp. 141–153. URL: http://people.freebsd.org/~jlemon/papers/kqueue.pdf (visited on 11/23/2012).

[Lie96] Jochen Liedtke. "µ-Kernels Must And Can Be Small". In: *5th International Workshop on Object Orientation in Operating Systems (IWOOOS)*. IEEE, 1996, pp. 152–155. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.79.2355&rep=rep1&type=pdf (visited on 01/23/2013).

[LW09] Adam Lackorzynski and Alexander Warg. "Taming subsystems: capabilities as universal resource access control in L4". In: *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*. IIES '09. 2009, pp. 25–30. URL: http://os.inf.tu-dresden.de/papers_ps/lackorzynskiwarg09_iies09-taming-subsys.pdf (visited on 03/02/2013).

[MLB] MLBA. *XDispatch*. URL: http://opensource.mlba-team.de/xdispatch/ (visited on 11/12/2012).

[Mol06] Ethan Mollick. "Establishing Moore's law". In: *Annals of the History of Computing* 28.3 (2006), pp. 62–75. URL: http://gigapaper.ir/Articles/Most_Downloaded_Papers_from_all_IEEE_Journals/Annals_of_the_History_of_Computing_IEEE/Establishing_Mooreapos;s_Law-Mollick.pdf (visited on 02/10/2013).

[Moo65] Gordon Earle Moore. "Cramming More Components onto Integrated Circuits". In: *Electronics* 38.8 (1965), pp. 114–117. URL: http://www.cs.utexas.edu/users/fussell/courses/cs352h/papers/moore.pdf (visited on 02/10/2013).

[RKC] Tord Romstad, Joona Kiiski, and Marco Costalba. *Stockfish—Powerful open source chess engine*. URL: http://stockfishchess.org/ (visited on 02/23/2013).

[Sta11] International Organization for Standardization (ISO). *ISO/IEC 14882:2011: Programming languages — C++*. Sept. 2011.

[Ste11] Daniel Steffen. *Mastering Grand Central Dispatch*. Presentation Slides. Apple Worldwide Developers Conference, June 2011. URL: https://developer.apple.com/devcenter/download.action?path=/wwdc_2011/adc_on_itunes__wwdc11_sessions__pdf/210_mastering_grand_central_dispatch.pdf (visited on 04/13/2013).

[Sut05] Herb Sutter. "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software". In: *Dr. Dobb's Journal* 30.3 (Mar. 2005). URL: http://www.gotw.ca/publications/concurrency-ddj.htm (visited on 02/20/2013).

[Teaa] Apple Inc./The GCD Team. *libdispatch—user space implementation of the Grand Central Dispatch API*. URL: https://libdispatch.macosforge.org/ (visited on 11/12/2012).

[Teab] The Clang Team. *Language Specification for Blocks*. URL: http://clang.llvm.org/docs/BlockLanguageSpec.html (visited on 02/14/2013).

A. Raw Benchmark Data

Slices	Threads	Minimum	Maximum	Average
1×1	1	3,685	3,687	3,686
1×1	2	3,653	3,655	3,654
1×1	3	3,653	3,655	3,654
1×1	4	3,647	3,649	3,648
1×1	8	3,647	3,649	3,648
1×1	10	3,653	3,655	3,654
1×1	12	3,647	3,649	3,648
2×2	1	3,684	3,686	3,685
2×2	2	2,254	2,275	2,267
2×2	3	1,589	1,591	1,590
2×2	4	1,589	1,591	1,590
2×2	8	1,589	1,591	1,590
2×2	10	1,604	1,606	1,605
2×2	12	1,589	1,591	1,590
4×4	1	3,660	3,662	3,661
4×4	2	2,431	2,478	2,454
4×4	3	1,963	2,048	1,997
4×4	4	1,488	1,538	1,521
4×4	8	1,522	1,542	1,531
4×4	10	1,374	1,407	1,383
4×4	12	1,500	1,506	1,503
6×6	1	3,680	3,681	3,680
6×6	2	2,562	2,627	2,602
6×6	3	1,895	1,928	1,915
6×6	4	1,590	1,621	1,611
6×6	8	1,569	1,614	1,594
6×6	10	1,681	1,711	1,699
6×6	12	1,575	1,616	1,599
8×8	1	3,614	3,616	3,615
8×8	2	2,497	2,502	2,500
8×8	3	2,020	2,061	2,046
8×8	4	1,553	1,583	1,569
8×8	8	1,539	1,579	1,557
8×8	10	1,580	1,614	1,597
8×8	12	1,494	1,524	1,510
10×10	1	3,687	3,689	3,688
10×10	2	2,561	2,585	2,573
10×10	3	1,971	1,983	1,976
10×10	4	1,737	1,784	1,762
10×10	8	1,606	1,637	1,626
10×10	10	1,597	1,623	1,613
10×10	12	1,506	1,536	1,522

Table A.2. *Primary Data—Fraktal, PThreads Version [L4Re]*

Slices	Blocks	Minimum	Maximum	Average
1×1	1	3,623	3,625	3,624
1×1	2	3,621	3,622	3,621
1×1	3	3,621	3,623	3,622
1×1	4	3,621	3,622	3,622
1×1	8	3,621	3,623	3,622
1×1	10	3,620	3,622	3,621
1×1	12	3,621	3,623	3,622
1×1	∞	3,621	3,623	3,622
2×2	1	3,622	3,624	3,623
2×2	2	2,252	2,253	2,252
2×2	3	2,260	2,305	2,283
2×2	4	1,954	2,171	2,054
2×2	8	1,578	1,738	1,631
2×2	10	1,773	2,033	1,859
2×2	12	1,578	1,809	1,664
2×2	∞	1,717	1,877	1,816
4×4	1	3,596	3,598	3,597
4×4	2	2,189	2,212	2,199
4×4	3	1,797	1,817	1,806
4×4	4	1,615	1,726	1,669
4×4	8	1,354	1,495	1,420
4×4	10	1,294	1,409	1,361
4×4	12	1,287	1,566	1,427
4×4	∞	1,073	1,223	1,161

Slices	Blocks	Minimum	Maximum	Average
6×6	1	3,613	3,615	3,614
6×6	2	2,154	2,176	2,163
6×6	3	1,647	1,701	1,667
6×6	4	1,636	1,783	1,675
6×6	8	1,431	1,525	1,481
6×6	10	1,390	1,598	1,467
6×6	12	1,226	1,338	1,292
6×6	∞	1,103	1,170	1,131
8×8	1	3,546	3,548	3,547
8×8	2	2,125	2,147	2,135
8×8	3	1,616	1,632	1,621
8×8	4	1,721	1,912	1,800
8×8	8	1,363	1,436	1,405
8×8	10	1,267	1,375	1,326
8×8	12	1,284	1,517	1,338
8×8	∞	1,020	1,350	1,179
10×10	1	3,615	3,616	3,616
10×10	2	2,176	2,177	2,176
10×10	3	1,623	1,636	1,627
10×10	4	1,632	1,868	1,700
10×10	8	1,423	1,468	1,451
10×10	10	1,383	1,444	1,409
10×10	12	1,340	1,404	1,380
10×10	∞	994	1,070	1,041

Table A.3. *Primary Data—Fraktal, GCD Version [L4Re]*

Slices	Blocks	Minimum	Maximum	Average	Slices	Blocks	Minimum	Maximum	Average
1×1	1	3,624	3,638	3,629	6×6	1	3,615	3,623	3,619
1×1	2	3,625	3,635	3,627	6×6	2	2,130	2,197	2,165
1×1	3	3,625	3,629	3,627	6×6	3	1,792	2,036	1,894
1×1	4	3,624	3,639	3,631	6×6	4	1,761	1,953	1,845
1×1	8	3,624	3,638	3,627	6×6	8	1,415	1,638	1,510
1×1	10	3,625	3,642	3,627	6×6	10	1,340	1,419	1,387
1×1	12	3,624	3,640	3,630	6×6	12	1,233	1,463	1,329
1×1	∞	3,624	3,638	3,629	6×6	∞	1,086	1,178	1,120
2×2	1	3,622	3,633	3,625	8×8	1	3,549	3,560	3,554
2×2	2	2,256	3,105	2,814	8×8	2	2,069	2,138	2,099
2×2	3	2,261	2,443	2,343	8×8	3	1,757	1,939	1,841
2×2	4	2,044	2,193	2,141	8×8	4	1,579	1,933	1,737
2×2	8	1,671	1,913	1,774	8×8	8	1,427	1,614	1,499
2×2	10	1,630	1,849	1,732	8×8	10	1,248	1,570	1,410
2×2	12	1,677	1,954	1,769	8×8	12	1,330	1,394	1,355
2×2	∞	1,587	2,038	1,768	8×8	∞	1,029	1,131	1,079
4×4	1	3,596	3,605	3,600	10×10	1	3,619	3,630	3,623
4×4	2	2,173	2,209	2,199	10×10	2	2,095	2,176	2,134
4×4	3	1,817	2,069	1,944	10×10	3	1,764	1,999	1,860
4×4	4	1,644	2,088	1,868	10×10	4	1,731	1,847	1,784
4×4	8	1,395	1,518	1,461	10×10	8	1,437	1,710	1,545
4×4	10	1,281	1,539	1,393	10×10	10	1,365	1,539	1,451
4×4	12	1,313	1,537	1,388	10×10	12	1,379	1,525	1,441
4×4	∞	1,225	1,362	1,273	10×10	∞	1,015	1,077	1,042

Table A.4. *Primary Data—Fraktal, GCD Version with Worker Migration [L4Re]*

Threads	Minimum	Maximum	Average
1	4,020	4,139	4,035
2	1,934	2,311	2,084
3	1,465	1,838	1,607
4	1,234	1,599	1,420
8	1,488	2,231	1,677
16	1,856	2,146	1,947
32	1,906	2,412	2,113

(a) *PThreads Version*

Blocks	Minimum	Maximum	Average
1	3,673	3,677	3,675
2	2,677	3,832	3,089
3	3,415	6,956	4,247
4	3,795	4,542	4,151
8	4,421	5,096	4,721
16	5,706	7,013	6,164
32	6,714	9,269	7,965

(b) *PThreads Version*

Table A.5. *Primary Data—Stockfish [GNU/Linux]*

Threads	Minimum	Maximum	Average
1	8,311	8,316	8,314
2	5,450	5,962	5,718
3	4,907	5,395	5,153
4	4,833	5,117	4,942
8	5,306	5,779	5,482
16	6,057	6,691	6,419
32	7,041	8,071	7,441

(a) *PThreads Version*

Blocks	Minimum	Maximum	Average
1	7,762	7,766	7,764
2	55,753	117,459	80,135
3	120,264	202,996	158,038
4	163,998	300,619	224,793
8	250,066	401,166	330,666
16	424,063	552,422	488,155
32	468,285	772,546	588,293

(b) *PThreads Version*

Table A.6. *Primary Data—Stockfish [L4Re]*

Blocks	Minimum	Maximum	Average
1	3,746	3,979	3,771
2	2,356	3,385	2,743
3	2,499	5,231	3,192
4	2,458	4,507	3,383
8	3,159	6,259	3,915
16	3,787	17,490	5,734
32	5,098	64,562	11,902

(a) *With Dispatch Queue Hierarchy*

Blocks	Minimum	Maximum	Average
1	16,833	16,936	16,915
2	12,512	20,973	15,333
3	22,803	49,741	41,484
4	27,693	88,092	53,093
8	52,029	271,172	146,935
16	110,925	326,576	211,622
32	208,242	585,088	368,139

(b) *With Dispatch Queue Hierarchy*

Table A.7. *Primary Data—Stockfish, GCD Version, Optimizations [GNU/Linux]*

Blocks	Minimum	Maximum	Average
1	5,124	10,056	5,679
2	7,184	25,951	10,870
3	9,265	22,781	12,652
4	10,113	24,235	15,997
8	12,204	28,813	20,166
16	21,566	49,655	33,528
32	48,892	84,037	69,165

Table A.8. *Primary Data—Stockfish, GCD Version, 200 Standby Workers [L4Re]*

B. License