# Implementing Pthreads on the XMOS Xcore

Research Review and Workplan

# Contents


1. Executive Summary
    1.1  Objectives and Deliverables
    1.2 Added Value
2. Introduction
    2.2 Motivation
    2.3 Scope
3 Parallel programming paradigms
    3.1 POSIX Threads
    3.2 OpenMP
    3.3 Higher Level Language Standard Libraries
    3.4 Assembly and Instruction Set level
4 The xCore, XC and the XS-1 Architecture
    4.1 xC
    4.3 XS-1 Instruction Set and Assembly
    4.4 Uses of the xCore
5 Parallel and embedded system benchmarking
    5.1 Comparison of different parallel programming paradigms
    5.2 Comparison of embedded systems & energy efficiency
    5.3 Comparison of parallel embedded systems
6. Workplan Breakdown
    6.1 Simple Threading (3 Weeks)
    6.2 Thread Stack Management (2 Weeks)
    6.3 Simple Mutex (2 Weeks)
    6.4 Pthread keys (2 Weeks)
    6.5 Testing (3 Weeks)
7. Risk Analysis
8. Implementation
    8.1 Pthreads specification
    8.2 The Xcore and XMOS technology
        8.2.1 Resources
        8.2.2 Channels
        8.2.3 Event Handling
        8.2.4 Scheduling and contention
        8.2.5 Process sharing
    8.3 Thread Creation and Lifecycle
        8.3.1 Thread Initialisation
        8.3.2 Thread Stack Space
        8.3.3 Thread Headers
        8.3.4 Thread Resources
        8.3.5 Thread Identifiers
        8.3.5 Synchronised threads

8.3.6 Thread Tasks
8.3.7 Thread Cleanup
8.3.8 Main
8.3 Thread Internal Functionality
8.3.1 Cleanup stack
8.3.2 Self
8.3.3 Cancellation status
8.3.3.1 Cancellation revectoring
8.3.3.2 Self Cancellation
8.4 Thread External Functionality
8.4.1 Joining
8.4.2 Cancellation
8.4.3 Keys
8.5 Synchronisation Primitives
8.5.1 Xcore Hardware Locks
8.5.2 Spinlocks
8.5.2.1 Spinlocks and Context Switching
8.5.3 Mutexes
8.4.3.1 First Implementation
8.5.3.2.Second Implementation
8.5.3.3 Mutex Event Handling
8.5.3 Barriers
8.5.3.1 Barrier Release
8.5.3.2 Barrier Continue
8.5.3.3 Synchronisers
8.5.4 Read/write locks
8.5.5 Condition Variables
8.5.5.1 Condition Variable Atomic Mutexes
8.5.5.2 Condition Variable Broadcast and Signal
8.5.5.3 Condition Variable Contact
8. Bibliography

# 1. Executive Summary

The XMOS Xcore is a multi-core parallel embedded processor architecture designed to enable rapid responses to inputs and allow data to be easily transferred between cores. As the internal architecture of the processor has several important differences from more standard designs, the two ways of writing code that takes full advantage of the multiple cores is either using a variant of C named XC developed by XMOS or through coding in the assembly language used by the processor.

On more standard architectures, there are many other options for developing parallelised code including the popular pthreads.h library used for POSIX thread control. Currently there is not a version of the pthreads.h library available for the Xcore architecture, which means that code has to be specifically written for the Xcore rather than using more universal functionality.

## 1.1 Objectives and Deliverables

This project aims to produce an implementation of the pthreads library, which is used to handle multithreading on POSIX systems in the standard C language, for the xCore.

A second part of the project will be to compare programs constructed using the pre-existing XMOS tools to programs using the new pthreads implementation in areas such as running speed, size and ease of development.

- An partial implementation of the pthreads library for the XMOS xCore, conforming to the relevant IEEE standard for the pthreads library.
- A testing system for comparing programs designed using purely the pre-existing tools made by XMOS for the Xcore to programs using the new pthreads library.
- An evaluation of the situations where porting pre-existing code using pthreads is more productive than rewriting the code in XC, weighing loses in performance vs simplicity and portability of code.

## 1.2 Added Value

The XMOS Xcore is a commercial product with a variety of uses and the potential ability to more easily take pre-existing code and transfer it to running on the Xcore architecture could be used as an added selling point. Depending on code complexity and the resources of a company, even if the code produced by using the new pthreads implementation is potentially not as efficient in execution as pure XC code or XS1 assembly, the time saved from rewriting code using the pthreads standard could be a benefit.

Designing a system to comparing various aspects of two programs is another potentially useful result, although much of the work towards this has potentially already been done in XMOS's standard development tools for the xCore. Whilst it is unlikely due to the specificity of this project that the tools here could be directly used for comparing programs designed for other systems, refining and expanding their uses could be an interesting area of future study.

# 2. Introduction

The XMOS Xcore is a specialised series of multiprocessors designed for use in embedded systems, featuring specialisations that allow for rapid responses to inputs - allowing substantially faster response times than comparable systems in both best and worst cases [1]. To enable this, XMOS designed the Xcore to function without many of the features that chips designed to work with operating systems use to control and manage concurrent and parallel processing, such as interrupts. XMOS also designed the Xcore with many features that are normally made in software inherently built into the processor's hardware which also helps keep response times to a minimum.

XMOS has a pre-existing development kit revolving around the XC language, which already features provisions for using the xCore's parallel processing capabilities which has been used for numerous projects already. Due to deeply embedded nature of the processor, assembly level programming is also fairly common, using the XS-1 instruction set XMOS designed for the xCore.

Pthreads is a C library that supports multithreading on POSIX systems with various similar implementations existing for other operating systems such as Windows. Pthreads is also used as part of the implementation of several other libraries that deal with parallel processing or multithreading, such as C++'s <thread> library. Pthreads is currently not implemented for the xCore, mainly since the Xcore is designed in such a way that an operating system is not necessary for functionality.

Currently, the only way to access the full parallel processing power of the Xcore is to either code and compile part of a program designed for the Xcore in xC, or to use XS-1 assembly language. This requires specific knowledge of a niche language and processor system, whereas pthreads is a much more universal system that many programmers will have a degree of experience with. A functional implementation of pthreads on the Xcore would therefore potentially save some development time or help users match XC functionality to that which they are more familiar with.

## 2.2 Motivation

Whilst it is unlikely that peak performance on any given platform will ever be achieved by cross-platform code, having the ability to relatively easily move code from one platform (in this case, from a POSIX system to the xCore) to another has several obvious benefits. The main benefit is saving potential programmers the learning time of having to immediately learn new programming paradigms when beginning development for a given system - a benefit of potential interest given the commercial purpose of the xCore.

Another benefit is it makes being able to test whether a given system is the optimal one for a given task without having to redesign the task's code. Even if the ported code only achieves performance below the peak performance, it can perhaps be used as a benchmark for what a complete implementation would perform like.

Another benefit specifically to pthreads is that many other higher level systems are implemented off the pthreads library. If a pthreads port is developed successfully, it would potentially pave the way for implementations of other systems such as C++'s (a language that can already be compiled on the xCore) <thread> library, which can used in different programming styles to the imperative programming style used by C or the event driven one used by xC.

Further on, developing a structured system for comparing different methods of implementing parallel processing for the Xcore would be of use not only for being able to see the performance costs of the implementation of the pthread library, but also for later developments. Whilst a lot of research has been done into methods for comparing different parallel programming paradigms in higher performance computing, the equivalent research for embedded systems is sparser. Often there are only a few methods of programming in parallel for a deeply embedded processor, which makes comparing them less important than larger systems.

## 2.3 Scope

Whilst there is a very large body of literature regarding parallel processing, embedded systems and the overlap between the two, the very specific nature of implementing a library for one system naturally limits the scope of the project.

The main language used will be C, with substantial sections programmed in XC and the occasional use of raw assembly language for accessing functionality such as thread creation that may not otherwise be exposed to the user.

Taking a conservative estimate for the difficulty in adapting pthreads functionality to the xCore, a focus will be one getting a very simple and limited version of pthreads produced first, to enable testing and prevent the project from delayed in an unworkable state. Later expansion will include mapping more niche pthreads functionality to more unusual elements of the Xcore architecture.

Once a pthreads system is able to be used to execute code, the next step will be judging the comparative performance to existing techniques and the relative ease of porting non-Xcore programs to the xCore. This will involve writing programs in both XC and C, as well as establishing a method for comparing the ease of porting programs.

# 3 Parallel programming paradigms

Parallel programming is a well-established area of study, with many cases where it can be the source of substantial increases in performance, even dating back to before most systems were parallelised [2]. As time has passed in Computer Science and processors have become smaller and easier to combine into multicore processors, even deeply embedded processors often have two or more separate processors.

Naturally, this has led to many different possible ways of implementing parallel code, all with various situations where they hold advantages in portability, performance and memory usage. One of the problems however is that there is far from a consensus on exactly which techniques are the best; and this lack of agreement causes numerous issues where different groups design software and hardware in different ways to meet the same end goals [3]. This plurality of options means that it is difficult for programmers to gain experience in parallel programming that can be carried between different projects, as the skills gained working on one project may be very different to those needed on another.

Despite this, there are several widely supported parallel programming paradigms used in computer science, that provide support for a wide variety of systems.

## 3.1 POSIX Threads

POSIX Threads, or pthreads, is C library that governs thread creation and control at a low level on POSIX compliant systems, such as those running Linux. As with all POSIX features, it is defined by the IEEE [4] which ensures that whilst the internal workings of the system may be different, to the programmer the behaviour of pthreads is as consistent across systems as possible.

Pthreads has several issues that prevent widespread usage. Firstly, there are slight performance overheads attached to it due to the common requirement of having a thread separate from the working threads controlling creation, destruction and timing compared to other paradigms such as OpenMP [5]. Secondly, as it is very low level code designed for an equally low level language, it is often more complex and unwieldy than needed by a solution. Finally, most POSIX compliant machines are clearly intended to be managed by an operating system, which for deeply embedded systems can consume valuable memory and processor time.

Despite these downsides, pthreads has a place as a tool that can be used for small adjustments to other higher level parallel programming paradigms in cases where there are small optimisations to be gained [6], and as one of the lowest level ways of interacting with threads. Indeed, some higher level threading systems will be build on top of a pthreads system, such as the <thread> library for C++.

## 3.2 OpenMP

OpenMP is a collection of compiler additions that allow higher level threading control for a variety of languages, mainly through the use of pragmas added into the code. Through being less precise about the exact mechanics of how threads are created and managed OpenMP can achieve slight speed increases in many situations over more specific systems like pthreads.

OpenMP is one of the most common methods of working with very large and powerful computer systems, such as those used in high performance computing and much of the literature regarding it reflects this [7]. Even so, it has been used to develop software for various smaller embedded systems successfully [8] and this flexibility contributes to its prominence in academia.

One of the developments in OpenMP during recent years has been the adaptations made by various groups to allow OpenMP to work on larger networks of parallel computers through combining in Message Passing Interface (MPI) structures and techniques.

## 3.3 Higher Level Language Standard Libraries

Most modern programming languages support elements of parallel computing, with the degree of support varying from language to language. Python is an example of a high level language with both threading and multiprocessing libraries, yet despite this, Python is also an example of a language with limited support for multithreading due to the Global Interpreter Lock, which prevents the level of multithreading possible with lower level languages such as C's pthreads [9].

Even in languages with no natural support for parallel processing, there are libraries like pthreads that support parallelisation. Often, a language's native parallel computing libraries resemble pthreads in structure, with similar functions and data objects [10]. A limiting factor on the use of language standard libraries is that they are constrained to devices that can run a given language; and many embedded systems have no support for high level languages.

The advantage to using a language's native tools is that they are often able to be intuitively combined with the normal programming techniques of the language. For example, object oriented languages often can threads as objects themselves and this allows programmers experiences with the language to use the techniques with limited training.

Some languages are designed from the ground up to support specific kinds of parallel programming, such as the XC language designed by XMOS for their proprietary

processor architecture. These languages are often incredibly well tuned for their specific situations, but naturally cannot be easily ported to different platforms and contribute to the bloat in possible parallel programming paradigms.

## 3.4 Assembly and Instruction Set level

Whilst it is almost never necessary for more powerful operating systems and can often introduce more problems than it solves, there is always the potential for programming multithreaded or multiprocess programs in raw assembly language.

Underneath the abstraction provided by libraries and other methods, most modern systems will have support for parallel programming available in the assembly language [11], even if the operating system is relied on for certain elements of scheduling and resource allocation. For systems where memory is limited and the programs loaded are simple enough that they can be easily designed in raw assembly, assembly coding can be a viable method for getting the maximum performance.

The xCore's assembly language, XS-1, has many commands relating to parallel programming, understandably due to how many of the features normally managed by the operating system or other software components are instead rendered in hardware. More conventional assembly languages and instruction sets will have fewer direct parallel programming commands as more of the burden of controlling programs will be offloaded to the operating systems.

# 4 The xCore, XC and the XS-1 Architecture

The Xcore is a processor designed for embedded systems, with up to 32 parallel cores per processor. Each core is able to execute and react to inputs independently of the others, and due to this, the system can have substantially faster reactions to inputs than comparable processors [1]. Xcore processors can also be combined into large systems easily, using ‘tiles’ to allow different processors to communicate.

Internally, the Xcore has several important differences from more standard processors. One notable difference is that the Xcore has no cache, which means that memory reading can be given a constant time rather than an estimate. The Xcore also has hardware implementations of functionality normally left for software to handle - such as task scheduling and message passing between cores [12]. This allows both faster and more predictable execution, both of which are desirable for fast response times and precise development.

The downsides of the Xcore are the expected drawbacks of a deeply embedded processor - limited memory and slower speeds when compared to larger computers. Despite this, there are many uses for the chip, ranging from audio controllers to robotics.

## 4.1 xC

XC is a variant of the C language, designed by XMOS to make full use of the inherent parallelism in the Xcore processor, and to help facilitate the event driven programming that the Xcore is often used for. XC is compatible with standard C, featuring the same functionality with a few key additions, such as the parallel block.

```
int main () {
	par {
		task1(0);
		task2(1);
		task3(3);
	}
	return 0;
}
```

Parallel blocks, shown here with the *par* keyword are sections of code that will be executed on several cores at once, and this forms the basic element of the XC system for distributing tasks. Arguably, these are a more natural way of expressing parallelism than the methods found in other parallel programming paradigms such as OpenMP, as it shows grouping of various parallel tasks. The program will wait for all tasks in a single *par* block to complete before continuing to further sections of the code.

Each individual task in XC has a separate set of memory that it can use, and the compiler uses extra techniques beyond those naturally inherent to the C language to identify race conditions, access to deallocated memory and other undesirable memory accesses. As part of this, pointers in XC must be specifically flagged as being unsafe to allow full C style usage, and XC possess a variety of language constructs (such as interfaces and channels) for using the hardware supported message passing systems to allow data to be passed between tasks.

XC also has language constructs that support the event driven programming that the Xcore is particularly suited to, allowing tasks to wait for specific inputs either from other tasks on the processor or external inputs.

## 4.3 XS-1 Instruction Set and Assembly

XS-1 is a 32 bit instruction set specifically developed for the Xcore architecture. Many of distinctive features of the XC language, such as channels and interfaces, are supported at an instruction set level, rather than existing as higher level objects. XS-1 also has functionality that more normally would be delegated to an operating system, such as task scheduling, which is the result of the Xcore being designed to functional without an operating system. XS-1 also supports the expected operations of a complete instruction set, such as maths operations, boolean comparisons and memory read/writes [13].

The assembly language used by the Xcore is a fairly straightforward mapping of the XS-1 instruction set into a more human readable form. There are also a few compile time directives that can be used when compile the program or testing it with the XMOS-provided analysis tools, such as xTime which uses the deterministic nature of the Xcore to provide estimates for the running speed of a program[13].

## 4.4 Uses of the xCore

The uses of the Xcore are varied; whilst XMOS advertises the chip on their website for audio controllers, many other companies have used the Xcore for a variety of products.

Robotics is a fairly common use of the xCore, such as those produced by Robugtix[1]. A fairly common design of robotics is having embedded processors to control individual limbs and attachments with a more powerful controlling processor; the rapid response of the Xcore to inputs from the main controller makes it a good fit for this task.

A more unconventional use is the use of the Xcore as part of a haptic ultrasound system; by focusing ultrasonic waves, a device can create the feeling a physical object in space. Xcore processors are used here to control the sound emitting system.

[1] http://www.robugtix.com/, where they use Xcore chips to control the movement of the individual legs of their spider robots.

# 5 Parallel and embedded system benchmarking

The variety of techniques for programming in parallel has lead to a wide variety of research into which methods are optimal in given situations. Whilst the lack of a clear consensus suggests that the field has a way to go yet before solving the issue of finding the perfect solution, this is not due to a lack of research.

The nature of the systems means that comparing high performance computers and small embedded processors are going to test very different aspects of the processor. Despite this, there is overlap even if direct parallels are hard to find between the two areas.

## 5.1 Comparison of different parallel programming paradigms

The main parallel programming paradigm that most researchers test against is OpenMP. This is partly due to the wide usage of OpenMP, which is available on most systems, and also the ability of OpenMP to adapt to the system it is on, as it only exists as a series of compiler directives rather relying on explicit calls by the programmer as pthreads does. Nonetheless it is worth noting that OpenMP displays substantial differences in performance between systems and compilers [15], which shows why sound benchmarking systems are important even for well developed systems like OpenMP.

The general principle of benchmarking parallel programs is fairly straightforward; dividing up a mathematically or computationally complex program into smaller independent tasks and comparing run times. A popular approach is matrix multiplication, as it naturally lends itself to numerous smaller tasks, though there are many varied tasks [16]. For most situations, the primary concern is in testing run speed as memory is a lesser concern for the larger systems that often are used for these comparisons, though often performance gains are matched by increased memory usage from the overhead associated with each thread or process [17]. One of the difficulties in mapping higher performance computing benchmarking techniques to smaller embedded systems is that the more limited memory makes the choice of tests slightly more restricted.

## 5.2 Comparison of embedded systems & energy efficiency

When benchmarking embedded systems, more of the focus is on the performance capabilities of the hardware rather than than the performance of the code developed on this. There are also other factors that play into the benchmarking, such as the energy consumption and dissipation of the processor [18] which is often a

consideration for embedded systems that will be running on a limited power supply and in tightly packed environments. Whilst often the most effective way of reducing energy consumption is simply improving the program to run faster, there can be more obscure bottlenecks and limiting factors[2].

The variable importance of energy efficiency is an example of the difficulty when benchmarking embedded systems. As embedded systems are often designed specifically for single purpose and need to be tuned for that purpose, individual tests may not provide a complete picture of a chip's capabilities. Often, a series of different tests will be needed, leading to invention of entire testing suites for embedded processors. MiBench is a fairly typical testing suite, featuring tests tuned for a wide variety of different purposes and with a reasonably long history of being used in both industry and academia [20].

## 5.3 Comparison of parallel embedded systems

Almost all modern embedded systems are now parallel in structure; as the speed of individual processor cores begins to reach a plateau adding additional cores becomes the most effective way of speeding up computers.

Benchmarking parallel embedded processors is slightly more involved than testing large computers; the limited memory means that the benchmarking program cannot be overly long and it cannot require a large memory footprint. Combining this with the difficulties mentioned above when testing embedded systems designed for specific purposes, and it is clear that testing parallel embedded systems is a more complex issue than simply designing a single test.

ParMiBench is a variant of the MiBench testing suite mentioned above, and features tests designed for parallel embedded systems rather than the single processor architectures that MiBench was designed for [21]. ParMiBench uses simple algorithms (in terms of code length) with relatively long run times to provide a good approximation for the performance increases possible as more cores are used in a processor and is sufficiently widely used that other testing suites are developed using ParMiBench as a starting point [18].

[2] E.g. cache missing and memory access can be large drains on power [19]. The Xcore architecture avoids the former of these by simply not having any caches and only reading straight from memory.

# 6. Workplan Breakdown

## 6.1 Simple Threading (3 Weeks)

Deliverable: The deliverable for this section is the ability to create threads and destroy threads, and to control whether the thread is running separately from the program or not.

The aim with the first section of work is to perform the simplest operations that will allow multithreaded programs to be developed; namely, creating, destroying and detaching threads. Much of this will be fairly simple as the Xcore has direct assembly instructions for several parts of this, but a fair amount of time will be needed to learn the specifics of the Xcore assembly language.

At the beginning of this section, there will likely be a bottleneck in development whilst setting up the systems for creating and destroying threads. Without these systems in place, later developments will have no real context in how they can work. After passing this bottleneck, other systems should be more independent, and there will be more possibility for overlapping tasks.

## 6.2 Thread Stack Management (2 Weeks)

Deliverable: For this section, the deliverable will be the ability to dynamically readjust the stack size of a given thread, and to get the memory address of the thread's stack.

Once threads are usable for the simplest of cases, the next task will be to enable allocating more or less memory as needed. Due to the limited memory of the xCore, standard POSIX memory management style operations may prove to be inefficient or unworkable, in which case an alternative will be needed and will need to be constructed.

Depending on the exact results of the first stage of development, it may be that many of the problems in this section will need to be addressed earlier. It may also be, due to the memory layout of the xCore, that the thread stack address is not a useful value.

## 6.3 Simple Mutex (2 Weeks)

Deliverable: The deliverable for this section is a mutex that can be created, destroyed, locked and unlocked to allow synchronisation between threads.

Before being able to develop multithreaded programs that share resources between each other, it is necessary to develop data synchronisation primitives that prevent race

conditions. Mutexes are one of the simplest of these, and the one with the most possible uses. There is also support in XS-1 for a lock type data structure, which may be wrappable into a pthreads compliant mutex or lock.

## 6.4 Pthread keys (2 Weeks)

Deliverable: The deliverable for this section will be an implementation of the pthreads key system for sharing data.

An aspirational goal, presuming that there have been no overrun tasks before it, is to implement the pthread keys system for controlling shared data between threads on the xCore. The Xcore architecture has several hardware and instruction set level systems for sharing data, and it would be an interesting area of the project to see if the pthreads system and the Xcore system can be made to work together.

However, this is a less important area of the project than the earlier sections of implementation, and can easily be removed if necessary. There is an extra week with no specifically allocated task that can also be used for catching up on any outstanding tasks.

## 6.5 Testing (3 Weeks)

Deliverable: By the end of this section, data will have been collected comparing the existing XC implementation of several programs to a pthreads version. Data will also be gathered about the ease of converting platform independent pthreads programs into ones compatible with the xCore.

Testing the program means testing several aspects. As well as ensuring that the implementation works correctly, it needs to be compared in speed to standard XC programs. XMOS has an internal tool for comparing run times, which is one method of testing - though it is potentially unreliable for testing more complex code. Another method of testing would be designing programs with long run times focusing on repeated uses of the implemented functions from pthreads.

As well as testing efficiency, a comparison of how easy it is to port code from pthreads to XC is also important, such as comparing whether there are major structural changes needed. Presuming that the standards are met, this should be possible - if the standards are not meet for a reason, this becomes harder to evaluate.


Research Project Timeline
Simple threading
pthread_create
pthread_exit
pthread_join
pthread_detach
Thread stack management
pthread stacksize
pthread stackaddr
Simple Mutex
Implement pthread_mutex_create
Implement pthread_mutex_destroy
Pthread keys
pthread_key_create
pthread_key_destroy
Testing and comparison
XTime testing
Side by side comparison
Testing rewriting difficulty
Write up
Report
Poster
Duration
Core Deliverable
Aspirational Deliverable
0
5
10
15
Weeks

# 7. Risk Analysis

| Risk | Likelihood | Severity | Contingency |
|---|---|---|---|
| Development reaches a major bottleneck, and the interlinked nature of the pthreads library makes it impossible to skip past | Moderate | Minor | The timings allocated are very generous, with space given so that the most core functionality should be finished even if certain problems prove insurmountable. |
| Pthreads implementation is substantially less efficient than XC code | Unlikely | Minor | It is expected that the pthreads implementation is slower than XC due to overheads that XC will avoid. Even so, identifying major performance bottlenecks is a useful result to record and can be seen a positive result. |
| Impossibility of meeting pthreads standard | Likely | Minor | Maintaining a record of an deviations and noting the reasoning behind them in the final results. |
| Limited memory availability prevents a pthreads implementation | Unlikely | Major | Designing a smaller implementation that misses certain functions. |
| XC and Xcore architecture requires substantial time to learn | Unlikely | Minor | The conservative estimates for deliverables mean that a reasonable amount of time can be lost during the beginning of the project without hindering overall success. |

# 8. Implementation

## 8.1 Pthreads specification

In total, the
The pthreads standard only defines the externally visible functionality of the implementation, but in doing so leaves many of the internal workings intentionally open to interpretation. An example of this is how the data structures specifying various attributes of threads and various synchronisation primitives are only required to be accessible through specific functions - whilst the simplest form is on most implementation simply a struct that can be accessed directly in C, this allows an implementation to any additional functionality, as well as all the standard benefits of forcing a more object-oriented design, such as type checking.

The pthreads specification itself also makes it difficult to design an implementation simply by one-by-one creating functions matching the requirements, as the implementational requirements of one function may change required behavior of another (the most prominent example being the interactions between the cancellation functions and numerous functions across the project). Despite this, it is possible to broadly separate the pthreads standard into four main categories of related functionality.

The first category, and the most important and widely used is thread creation.  Aside from the obvious reason that without the ability to create threads, a multithreading library is useless, many programs that use pthreads only use it for creating threads.

The second category are functions used by threads to control their own state, such as adjusting their cleanup routines or cancellation state. In the current implementation these are very simple, as they can broadly be treated as serial threaded functions.

Thirdly, there are many functions relating to ensuring synchronisation between threads, providing means to prevent race conditions. These functions make up the majority of the pthreads header, but many of them can be reduced down to very similar underlying implementations.

A final category of functions are those that affect the global state of the program, an example being those that actually cancel a thread

## 8.2 The Xcore and XMOS technology

The Xcore features several features beyond what is supported by standard C, many of which have been required as part of this project. As well as the channel communication system, there has been need for the event handling, timers and thread creation which have all required understanding the hardware and the assembly instructions needed to use them.

### 8.2.1 Resources

The Xcore has a number of hardware resources that can be allocated. Timers, threads, channels, synchronisers and locks are the five allocatable resource types. Ports are also considered resources, but are not allocated in the same way.

Resources are allocated with a *getr* instruction and can be freed with a *freer* instruction, the former giving a 32 bit identifier and the latter requiring an identifier. In XC the compiler manages resources through having predefined types for each one, but in C, they must be manually managed.

Channels and synchronisers have additional requirements due to being associated with specific threads; once allocated only the relevant thread can use them without causing an exception. They also require that they are not in use before being freed - either having all their threads joined or all data read out.

### 8.2.2 Channels

The Xcore channel communication system works on a fairly limited core set of instructions. The *in* and *int* (for single bytes) allow data to be read into a local register, and *out* and *outt* allow a channel to send data to another destination channel. The architecture essentially requires that these are used in matching pairs of sending and receiving, as channels where data is unread cannot be freed, and sending data to a channel which is full causes the instruction to pause until the channel has been emptied.

If data is requested from a channel, and the channel does not have enough to fulfill the request, the instruction pauses until the channel has data sent. This formed the basis of the initial implementations of various synchronisation tools for the project, as it provides the most straightforward way of pausing threads.

As well as direct data, channels also allow control tokens to be sent using similar commands. The most important aspect of control tokens is that channels require a specific CT_END control token to be sent or recieved before they are able to be freed. Often, especially in early iterations of the project, channels would wait for a signal

consisting purely of a CT_END control token, as they were using channels for pausing rather than data transfer.

### 8.2.3 Event Handling

A major selling point of the Xcore is the ability to rapidly respond to inputs, facilitated by the hardware level events system. Timers and channels can be configured to have events enabled and given a vector, after which if they are triggered - by data being sent on a channel, for example - the relevant thread is set to run from the vector. This is typically much faster than operating system level interrupts for other systems, and normally happens in up to two processor cycles.

Event enabling can be done in both synchronous style where a thread is given a *waiteu* or similar instruction, which pauses a thread until an event occurs. Alternatively, events can be enabled asynchronously so that the thread can continue running unless an event happens; to do this, the event enabled bit of the thread's status register has to be enabled.

XC is the typical method by which  most programs using this are designed, where there is a *select*{} construction similar to a standard C switch which allows the compiler to handle the exact underlying logic. However, several elements of the project handle events unusually, which required inline assembly code to function correctly.

### 8.2.4 Scheduling and contention

The Xcore features hardware level scheduling for threads, allowing them to evenly share processor time. One major selling point of the Xcore systems is that it ensures that thread performance degrades steadily and consistently - each thread is given one instruction in turn over each processor cycle. As this is implemented in hardware, it is impossible to directly adjust beyond a few additional options such as 'fast' scheduling where threads maintain a position in the schedule when paused (which ensures that the two processor cycle maximum for waking up for events instead becomes one).

However, one assumption of the POSIX operating system is that the operating system has much more control over the scheduling. An example of this is the ability to offer round robin scheduling, where threads execute for a defined period without interruption before stopping and allowing other threads to continue. This is impossible on the Xcore processor as there is no operating system to oversee it - even with an operating system, the events system would make it difficult to stop a thread without simultaneously losing that thread's program counter information.

Contention is a system for allowing an operating system to know how to handle threads seeking to use processor resources, with the two pthreads standard options of system wide or process wide contention. This falls into the same problems as scheduling - as it is done by the hardware rather than software as on large POSIX systems, it is not modifiable by the xthreads implementation.

### 8.2.5 Process sharing

POSIX systems are designed with the intention of having numerous processes active that normally do not directly interface, but there are allowances in the pthreads standard for sharing certain sections of data between threads, such as synchronisation primitives. Whilst this would be useful on larger systems, the Xcore is used for single processes, meaning that process sharing has no purpose. As such, process sharing has been left unimplemented.

## 8.3 Thread Creation and Lifecycle

The single most important functionality of pthreads is the thread creation, and this was the first part of the standard implemented in this project.

### 8.3.1 Thread Initialisation

Initialising a thread on the Xcore is a fairly straightforward process, as almost all of the operations are handled at the processor level rather than requiring further implementation.

Each Xcore maintains a collection of threads that are allocated like other resources.. When allocated, threads have undefined values for their stack pointer and program counter, which must therefore also be initialised to relevant values before the thread is allowed to run.

Despite the common misconception where it is believed that detaching or joining a thread is part of the procedure for starting a thread, thread execution actually begins during creation. It is advisable however that threads are joined or detached, as the alternative is to leave threads in an active state for the duration of the program.

### 8.3.2 Thread Stack Space

An unwritten assumption of pthreads is that the system running pthreads is allocating memory for each thread at creation time. On a larger processor, this is entirely possible to do, as the operating system will be able to manage the memory available to it mcuh better, and fragmentation will not become a major issue in most situations.

On an embedded system, memory is much scarcer which prevents this design from being viable. Instead, this implementation uses a statically allocated memory space that can be reused by threads over the runtime of the program. Whilst this would normally place a hard limit on how many threads can be active at once, the Xcore naturally only supports a specific number of threads rather than the more open-ended number that a larger processor would be able to.

Managing this additional space was predicted during development to be one of the biggest additional overheads compared to standard XC multithreading, as XC multithreading can more reliably offload the overhead of managing this to the compiler. In XC, the compiler deterministically knows how many threads are active at any given point and can decide stack addresses and sizes from that.

### 8.3.3 Thread Headers

As well as stack space, there is a need for various other pieces of data for each thread, such as the join status. These are stored adjacent to the stack space in memory.

### 8.3.4 Thread Resources

Typically in XC, resources like channels are only allocated when necessary - normally at the start of functions. In XC, just as in pthreads, each thread is created to run a specific function however, which means that often a thread holds a channel or other resources for its entire lifespan.

In the xthreads implementation, threads are each assigned two channels during their creation. The first is the thread channel which is used for purposes such as communicating with synchronisation primitives or other threads. The other is the cancellation channel which is used only for sending the thread a cancellation signal causing it to cancel.

Because these resources are allocated for substantial periods of time, it is important to allow programmers to use the thread channel to send signals, and these additions to the pthreads standard are discussed later further. This is potential the closest compromise between the XC's native architecture and the pthreads standard for being able to use the Xcore's unique features.

### 8.3.5 Thread Identifiers

As the pthreads standard assumes that the thread stack is dynamically allocated from memory during the runtime of the program, it implicitly assumes that thread identifiers, either received during thread creation or from transferred data from using pthreads_self() are valid for essentially the run time of the program.

As the xthreads implementation uses a more static memory system, eventually thread identifiers will become invalid and refer to a thread that no longer exists at all in memory. To solve this, every time a thread ends, it increments a counter variable in that thread header. The composite thread id that is received from calls into the xthreads implementation is then given as:

$$\text{Composite identifier} = (\text{count} * \text{number of threads}) + \text{thread header index}$$

This means that it is possible to check whether a thread identifier is still valid by splitting the composite thread identifier into the count and the thread header

> Thread header index  = thread identifier % maximum number of threads
> Count = (thread identifier - thread header index) / maximum number of threads

If the count variable stored at in the thread header at the thread header index doesn't match, the implementation can tell that the thread identifier is outdated and correctly handle the situation.

### 8.3.5 Synchronised threads

The Xcore has two different types of threads - synchronised and unsynchronised - which differ in a few key ways. Synchronised threads are bound to a synchroniser, that allows the thread owning the synchroniser to control all the child threads attached at once.

Synchronised threads can be synced, where they wait until all threads reach a *ssync* command before being released by a *msync* command from the parent thread. Child threads currently performing a *ssync* command can also be terminated by the parent performing a *mjoin* command.

Despite the similarities to the pthreads standard, there are complications that actually prevent synchronised threads from being used in a pthreads compliant way, the most important being that synchronised threads cannot self-terminate, as required by the pthreads standard. Instead, they must be joined by the master thread - which is not guaranteed in a pthreads program, where any thread can join any thread. In a standard XC program, threads actually fall into a very strict hierarchy which allows parent and child threads to interact in this manner. Additionally, whilst a thread in pthreads can be marked as detached, which means that it can only self-terminate, threads created in a joinable can be later required to detach.

Several techniques were explored to allow synchronised threads to be incorporated into the implementation. The first attempt was to create synchronised threads, and check whenever a thread creates a new thread whether there were loose synchronised threads waiting to be terminated. This runs into a potential problem where the program could fail from lack of available threads if threads create a number of synchronised threads and then never check to clear them.

Another attempt was to create detached threads for each thread at thread creation, but then replace them with detached threads if they are later detached. The issue here that prevented this solution is that retrieving all the required data for a thread - namely the program counter, the stack address and other register stored values - is impossible or very difficult when the Xcore is running normall.

A possible work around to this would be to use the native debugging system, which does allow the extraction of register values from threads, but this would require rewritting the entire debugging system around this. Another seemingly possible work around would be to have the synchronised thread spawn a new unsynchronised thread, which would give it access to some of the values needed to clone the thread. However, any system that changes the program counter will prevent a proper clone thread being created, and without changing the program counter there is no way to make a thread execute a given routine.

A potential extension to the xthreads system would be a pthreads style function for creating synchronised threads whilst ensuring that the parent thread can join and terminate them. However, this is very similar in structure to existing XC parallelisation and has been decided to be low priority.

### 8.3.6 Thread Tasks

Once a thread has been created initialised and started, it cannot simply begin at the desired start routine, as work needs to done before the thread can begin functioning, namely allocating the thread's resources. To achieve this, the thread starts at a special xthreads_task() function that wraps around the target routine.

This also allows the thread to clean up after itself easily, as once the target routine completes and returns, the thread resumes inside the xthreads_task() function.

### 8.3.7 Thread Cleanup

Once a thread has finished its task or been cancelled, and any pending joins are completed if necessary, the thread begins the cleanup routine. This involves running through the stack of cleanup functions that have been specified, as well as the cleanup needed for any of the pthread keys that are currently active.

Additionally, this is where the thread bound resources are freed - namely the thread channel and cancellation channel. The implementation ensures that unless certain functions are misused (mainly the additional ones that go beyond the pthreads standard), this should be error free.

During development, incorrectly functioning code would typically be detected here, as extraneous data being sent to channels causes an exception when trying to free them, which should now be an impossibility.

### 8.3.8 Main

In the pthreads standard, there is no distinction made between the thread in which main is first invoked, and the threads spawned later. However, the compiler has control over the main thread, and it is against the standard (and realistically expected

programming practices) to require programmers to start main with a specific function call to set everything up.

Instead, a custom function is created and given the constructor attribute so that the compiler knows that it needs to be executed before main. This function sets the main thread up with various attributes to make it perform as if it were a thread created in the standard pthreads fashion.

However, there are certain consequences to this that are in a grey area according to the standard. Firstly, it is technically possible to perform a join on the main thread which causes any thread calling it to stall for the runtime of the program. It is also possible to try to cancel the main thread, which results in the program exiting with an exception.

## 8.3 Thread Internal Functionality

Each thread in a pthreads program has certain attributes that are purely adjustable and/or readable within the scope of that thread, such as the cleanup stack. Many of these functions are relatively straightforward, as they do no need synchronisation protection from other threads given their internal nature.

### 8.3.1 Cleanup stack

Each pthreads thread possesses a cleanup stack of function-argument pairs that get called during that thread's cleanup. The most obvious use of this is to free allocated memory in threads that may get cancelled before being able to free it normally, but there are occasional other uses.

This is implemented as a fairly straightforward stack in C, using malloc to create new nodes on the stack as calls are made. Typically embedded systems do not make much use of allocated memory due to fragmentation and scarcity issues, but most pthreads programs for embedded systems will be using a fixed number of non-ending threads and as they will not be using allocated memory, they will probably not require numerous cancellation functions.

### 8.3.2 Self

There are many situations where a thread needs to be able to access its own identifier, including inside many other pthreads functions. The identifier available from the pthreads style xthreads_self() function is the composite identifier that can be checked to ensure it's referring to a live thread.

The current implementation is a fairly straightforward search of the thread headers, trying to match the hardware id a thread can acquire during runtime (given using a *get id* instruction) to the saved hardware id in the thread header. This is less than ideal

however, as it gives a comparatively long run time to the function, which is needed fairly often in the underlying implementation.

An alternative technique that was experimented with was using one of the specialised registers that each Xcore thread possesses to permanently store the address of the relevant thread header. The registers used for kernel interactions - the kernel address and kernel stack pointer registers - were chosen because in the xthreads implementation, they were unused.

However, it proved unexpectedly difficult to get this to work smoothly. It is possible that there are other functions that rely on the kernel to work properly, as the problems occurred elsewhere in the program rather than in the xthreads_self() function.

### 8.3.3 Cancellation status

Whilst threads naturally can be cancelled from other threads, threads can only adjust their own cancellation status. Cancellation status consists of two values; the type, which is when a thread can be cancelled, and the state which is whether the thread can be cancelled.

Deferred cancellation, when the thread is only cancelled at specific points, is relatively straightforward to manage; the cancellation channel is set up with a vector at thread creation, and cancellation points are typically managed through a non-blocking event check. There are however various points in the program where functions that pause execution are also defined as cancellation points (e.g. xthreads_join()) and the event handling here becomes more complex.

Asynchronous cancellation gives no restriction on when a thread can be cancelled, though in practice there are many occasions where cancellation needs to be temporarily disabled. Using the Xcore's event system for asynchronous handling requires setting the event enabled bit in the thread's status register to one, which means that when a signal is sent to the cancellation channel, the thread will respond to the event on the next processor cycle.

Disabling cancellation is straightforward. To disable events in both deferred and asynchronous types, both the status register's event enable value must be set to zero (to disable asynchronous cancellation), and the events on the cancellation channel must be disabled (to ensure that cancellation test points do not pick up events). Disabling events on a resource does not automatically reset the event vector, which allows cancellation to be enabled again later without needing to revector every time.

#### 8.3.3.1 Cancellation revectoring

Certain situations require that the cancellation channel vector be adjust (an example is in the mutex event handling discussion). As part of these functions, the cancellation

channel needs to have its vector reset to the default state, otherwise there will be numerous unintended consequences.

#### 8.3.3.2 Self Cancellation

Pthreads threads can also cancel themselves using the pthreads_exit() function, which is handled different as there is no need to use the event system - though in theory a thread can indeed use the pthreads_cancel() function to cancel themselves as channels can send messages to themselves.

Cancellation here is simply calling the function that makes up the penultimate section of the thread task system - this causes the thread to act as if it had returned naturally from the routine it had been given.

## 8.4 Thread External Functionality

Simple programs where threads have no need to interact with the wider program are straightforward to design and program, as it is the interactions between multiple threads that add the complexity that parallel programming is known for. In standard XC, the compiler can handle much of the complexity through the abstraction of *par{}* blocks, but in pthreads style programming, the designer has to manage it themselves.

The pthreads standard also requires a few extra bits of global control and data, which have been grouped into this category.

### 8.4.1 Joining

Joining threads is an integral part of multithreaded programming, as it is the only way to be sure that a thread has finished - request a thread's cancellation may, depending on the state of the program, not actually get carried out immediately.

In the xthreads implementation, joining is achieved using channel communication, as threads that either waiting to join or to be joined can be safely paused whilst they wait for the other half of the system to reach the correct state. The difficulty with joining in the current implementation is preventing both threads from pausing waiting for signals from each other.

Each thread is given space in its thread header data for a return channel - the channel that it should transmit its return value (in the pthreads standard, the return value is expected as a void pointer, which matches the 32-bit token that a channel can send), which is initialised to a predefined flag at thread creation. When another thread (thread B) later attempts to join with the thread (thread A), it checks whether the value here has been changed; if the value is still at the default, then thread A is presumed to be still running and thread B can simply store its own channel into that slot, before waiting to receive a signal on its own thread channel.

When thread A later reaches the end of its execution, it checks the return channel value. If the value does not match the predefined flag indicating that there is no thread currently waiting to join, then thread A can simply pass its return value to the thread B through the channel stored as the return channel before continuing with the thread destruction process.

If thread A finished before thread B joins it, instead thread A pauses and waits for a signal on its thread channel from thread B when thread B reaches the join. In this case, there is a different predefined flag for thread A to set as the return channel value so that thread B can correctly identify that it needs to wake thread A up.

### 8.4.2 Cancellation

Cancelling another thread is a simple operation, it is only a matter of sending a single signal - even a CT_END token - to the cancellation channel of the thread in question. Multiple requests to cancel have the potential to cause undefined behaviour (typically either the call to xthreads_cancel() or the thread termination causing an exception), but this is an accepted cost within the pthreads standard.

### 8.4.3 Keys

Pthreads has a system for providing ‘keys’, which are data pointers with attached cleanup functions created automatically for each thread in a process. The main use of them is in applications where each thread needs an individual allocation of memory than can be automatically set up to be freed and allocated. This is not likely to be a major feature of many programs for the Xcore, but it has been included for completeness.

The implementation has space for four keys which are stored in two separate places. There is a globally accessible storage for the destructor functions which also serve as the flags showing whether a given key is in use or not - null destruction functions indicate that the key is currently inactive, whilst there is a defined flag for keys that are in use, but do not have specific destructor functions. Each thread header has storage space for the four keys available, which are initialised to null on each thread creation.

## 8.5 Synchronisation Primitives

### 8.5.1 Xcore Hardware Locks

The Xcore has native synchronisation primitives in the form of a small number of allocatable hardware locks. These allow locking and unlocking with single processor instructions, and naturally are very fast due to being implemented in hardware rather than software.

However, they are limited in number, with most Xcore tiles only having four, and can be difficult to share between multiple tiles. More critically, they don't directly match any pthreads synchronisation primitive directly; the closest comparison being mutexes, but there is no way to perform a timed wait on a hardware lock as the lock command blocks until completed.

As there already is a header file distributed by XMOS for accessing hardware locks, and the incompatibilities with the pthreads standard, the pthreads implementation does not interact with them outside of using one to protect the thread headers during thread creation.

### 8.5.2 Spinlocks

The simplest synchronisation primitive is the spinlock, which requires very little Xcore specific work. Whilst the Xcore does not have atomic operations such as compare-and-swap like those that mainstream architectures like x86 have, the cacheless nature of the Xcore's memory access means that it is possible to imitate the necessary functionality.

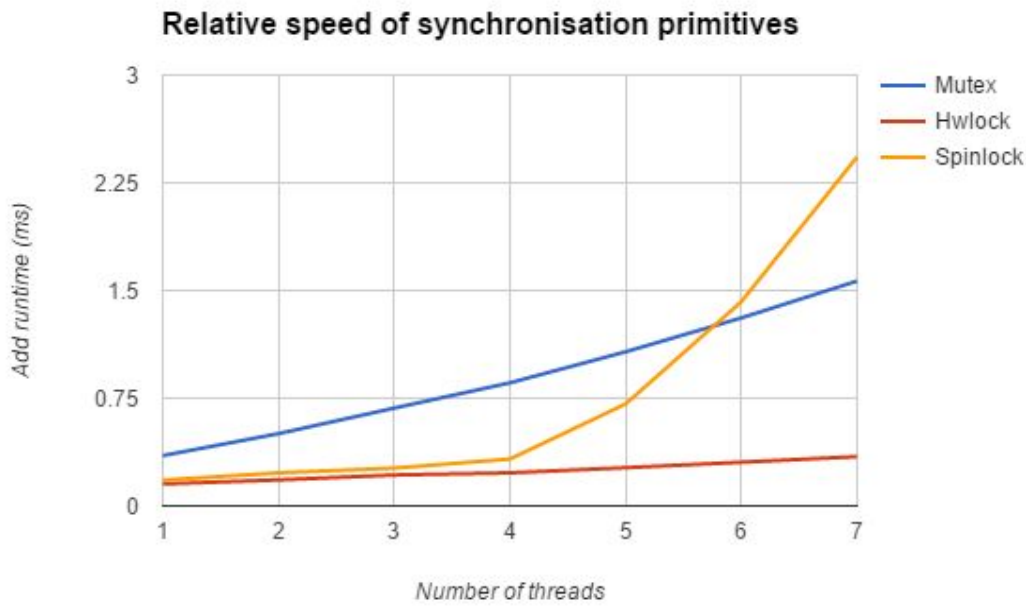
To lock a spinlock, a thread goes through three actions. Firstly, it reads from memory the value currently contained in the spinlock - if this doesn't match a predefined XTHREADS_NOTHREAD value, then the spinlock is unclaimed or uninitialised. If this check succeeds, the thread then attempts to write in it's own thread id, before waiting. After waiting, the thread then reloads the value from memory and compares it to the expected value.

Unlike on cached memory systems, the Xcore's memory access operations take close to consistent lengths of time, which means that this process consistently only allows one thread to continue holding the spinlock.

Pthreads defines two methods for locking a spin lock, trylock() which only makes one attempt and returns either a success or a fail, and the standard lock() which continues to 'spin' until a lock is acquired.

#### 8.5.2.1 Spinlocks and Context Switching

Conventional parallel programming guides will tell programmers to avoid using spinlocks to protect large blocks of functionality, as threads that are spinning attempting to lock a spinlock continue to consume processor time. Not only is this a problem in terms of program execution speed, as on standard systems context switching between active threads requires additional processor time, but for embedded systems, this continues to consume energy.




Xcore hardware locks and more advanced synchronisation primitives utilise hardware level systems for putting threads into a paused state, which prevents this unnecessary processor usage. However, the added benefits of this for processor speed are only apparent when the context switching comes into play; in the case of the XMOS StartKit the data for the graph above, this is when five threads are being run on its four cores. Energy consumption is harder to measure with the tools available, but shorter running time typically corresponds to lower energy consumption.

### 8.5.3 Mutexes

The core synchronisation primitive upon which many of the others depend is the mutex, which functions similarly in program flow control to a spinlock, whilst also allowing threads to be paused whilst waiting.

Both the following implementations share one common feature, in that they all have to use a spin lock to force threads into a serial pattern before adding them into the underlying data structure. If threads were able to add themselves into the data structure in a non-serialised way, then the mutex would be vulnerable to the same race conditions that it is intended to prevent. Both implementations are also pthreads compliant, though the earlier one has been unmaintained as the project has developed.

#### 8.4.3.1 First Implementation

The initial implementation made for mutexes was a simple queue in memory, where threads could add themselves when they needed to wait on the mutex.The thread unlocking a mutex can then get the top of the queue to find whatever thread needed to be woken up and send it a signal to wake up.

Threads that wait on the mutex are paused and are woken up by having a signal passed through their channel. Threads that are cancelled or are performing timed waits that time out simply mark their slot with a specific flag that means when unlocking the mutex, skip this space until you find a non-cancelled thread.

Whilst tgus was a working implementation and met all the requirements, there were several problems with this design that meant it was rejected and another implementation was designed.

The first problem is that it relied heavily on memory access, on a system with limited memory, as a Xcore tile typical only has a few megabytes of storage. Each mutex requires an integer per maximum thread to ensure that it has the capacity to be not fail from running out memory, which is a potential cost that can be avoided.

A related problem is that implementing mutexes in this way also tightly ties mutexes to a specific limit of threads. The Xcore architecture does only allow for a limited number of threads per processor so this is potentially mangagable, but the Xcore is also designed to work in larger configurations where the maximum number of threads may be much larger than expected from a single tile.

The final problem is that this design would place much of the processing burden onto the thread unlocking a mutex, and that burden would be erratically variable if threads are removing themselves from the queue. Both of these are traits that are unwanted where possible in the design, and as such a second implementation was designed to fix these mistakes.

#### 8.5.3.2.Second Implementation

The second implementation sought to take more advantage of the Xcore’s unique features rather than approaching the problem as one solvable in standard C design patterns.

Instead of utiliising a queue constructed in RAM, this design attempted to use threads and channels to construct a queue-like system. This system has some similarities to a one-directional linked list, and because of this can be scaled to accommodate any number of threads, whilst only requiring a small amount of RAM memory space as only the queue length and top of the queue need to be stored.

After locking the spinlock, a thread attempting to use this style of mutex reads the channel stored as the top of the thread queue and sets that as its destination, and then overwrites the top of the thread queue with its own channel. This means that data is stored instead in the thread's registers rather than RAM, which is more efficient, and this process is repeated for each additional thread added into the mutex's queue.

Each thread also keeps a copy of the count of waiting threads in its registers from the time of locking. This acts as its position in the queue, and allows the thread to recognise when it has become the thread waiting at the front of the queue, by checking when it's position equals one. Because the amount of data needed is sufficiently small, and threads waiting on the mutex are paused, the destination thread and the position can both be stored in the relevant thread's registers.

Once a thread has gathered all the data, it waits on it's channel until a signal is received. Each time a thread receives a signal, it reduces its position count by one, as it knows that a thread further along the chain has stopped waiting on the mutex. If a thread recieves a signal, but it does not become the thread holding the mutex, it echoes the signal along through its channel to the next thread. In this way, signals are passed along the queue ensuring that all the relevant threads are communicated with.

Each signal passed through the thread queue consists of two items of data; an integer sized data token and a single byte. For unlocking, both these will be zero, but it is necessary to be able to repair the queue if a thread leaves the mutex queue. The ability to pass different signals - normally taking the place of the channel address - allows much of the code for mutexes to be reused in other synchronisation primitives. This helps prevent the implementation becoming larger than necessary.

An interesting quirk of this system that is somewhat contrary to standard program procedures is that to send a signal down the thread queue inside a mutex requires locking the mutex's spinlock. Once the signal has been recieved by the relevant thread (e.g. the thread at the front of the queue if the signal is unlocking the mutex), it is that thread that then unlocks the spinlock, even though the spinlock is nominally held by another thread. This ensures that only one signal is ever being passed at once along the queue, and it ensures that the queue is not modified by threads being added when a signal is being transmitted.

When a thread quits the mutex, it sends a signal consisting of the channel it was previously targeting and the position that it was occupying. When this signal is passed to the thread that has been pushed over into the space the now-cancelled thread was occupying, the thread will adjust its destination to that sent in the signal. This ensures that the mutex remains usable as threads cancel.

#### 8.5.3.3 Mutex Event Handling

During the early stages of the project before a complete knowledge of the Xcore system had been developed, event handling was done using the channel communication operations, as they block unless the channel has received data. However, this raises several problems as further features are added in.

Firstly, and most importantly, is the issue of timed locks. Mutexes, read/write locks and condition variables all require support for timed waiting, where a thread can be instructed to wait for a given amount of time before returning a failure code if a synchronisation primitive cannot be locked.

To do this successfully, it is necessary to take advantage of the Xcore's event handling system. When performing a timed wait, instead of simply waiting on the channel directly, instead the channel, and the relevant timer, are given vectors to relevant sections of code and the thread waits on either of them to trigger before continuing.

The real complexity comes into play here when handling the signal echoing needed to pass information along the queue, when threads are also being cancelled at the same time. For example, if the thread one from the beginning (i.e. at position 2) cancels during a call to unlock the mutex, the timer will trigger the time-out subroutine before a message can be passed through to the thread that needs to be activated (i.e. the one at position 1).

The solution to this is to have a system where threads drop into a second inner event handling routine as part of the time out system. One trick of the Xcore architecture that is useful here is the event triggered by timers reaching a defined time is not cleared after it is handled, allowing the timer to be reused in this inner event loop.

However, unlike in the main event loop, if a blocking wait on events is used in the inner loop the overall performance of the mutex is compromised. Either it would be necessary to disable the timer if the mutex's spinlock is active before waiting for events and re-enabling it after the channel communication event had been performed, or the timer event would take priority and it would cause an infinite loop.

Instead, the implementation disables events for the timer once the timer event has been handled, before making a single attempt to lock the spin lock inside the mutex. If this succeeds, then the thread can proceed to exit the mutex queue as normal. If the spinlock is already claimed, then it does a non-blocking check of events by setting then clearing the thread's event enabled bit in the status register. If no events are trigger (e.g. there's a signal being passed but it has already been transmitted through this thread), the thread returns to the start of the timer cancellation handling routine.

A similar system also has to exist to allow cancellation requests to not damage the mutex's state. An added complication for cancellation is that channel events get cleared after they are triggered, unlike timer events, which requires some unconventional programming. Just as with the timer handling, cancellation handling uses a non-blocking system for checking for events, with the additional of sending a signal from the cancel channel to the cancel channel - typically a non-ideal situation.

However, since the signal is immediately handled if the event check doesn't trigger anything, and is needed to ensure that the thread can return to the cancellation handler if it has to be temporarily set to transmit messages. There is the potential for undefined behavior if the thread is cancelled several times in a row because the cancellation channel will become full and block further transmissions, but this is an accepted situation in the pthreads standard.

Ideally, event handling like this would be done in XC, where the compiler is able to understand the flow of control and optimise around it. However, because of the looping and unusual design needed, along with the need to include various xthreads headers that use non-XC compatible techniques like void pointers, this has had to be coded in a combination of C and assembly. A major challenge for this has been reconciling the compiler's attempts at optimisation with the event handling.

### 8.5.3 Barriers

In the pthreads standard, barriers are a synchronisation primitive used to ensure that threads only continue past a point once a given number of threads have reached it. The current implementation is actually completely derived from the mutex, as an additional pair of flags have been added to the mutex implementation covering two possible cases of barriers.

#### 8.5.3.1 Barrier Release

The barrier release flag causes each thread to retransmit the signal before immediately releasing themselves from the barrier, which makes the barrier release threads in a first-in last-out style. This is the most obvious way of implementing barriers, but it raises a potential problem in that the intuitive, if not strictly required, understanding of barriers would be that the first thread to reach the barrier would be the first one to continue once the barrier has been reached. Despite this potential awkwardness, this is the underlying implementation used as it is slightly faster than the 'continue' method.

#### 8.5.3.2 Barrier Continue

An alternative implementation that was developed but ultimately left unused was where the signal is passed along the thread queue until it reaches the final thread, and the final thread then resends the signal down the thread queue, and so on until all the threads are unlocked.

This ensures that threads leave the barrier in the order that they reached, but is substantially slower - if the ‘release’ method is O(n) in runtime, this one is $O((n^2 + n)/2)$. The pthreads standard does not specifically require that threads are released from a barrier in a specific order, so the faster method was used.

#### 8.5.3.3 Synchronisers

An alternative implementation would be to use synchronised threads and the synchronisation systems that the Xcore natively possesses to replicate the functionality of barriers. The combination of *ssync* and *msync* commands perform the same task, but the reasons discussed in 8.3.1, synchronised threads are not an option for this pthreads implementation.

### 8.5.4 Read/write locks

Read write locks are a variant on mutexes that allow programs to let certain operations that need synchronisation (typically memory writes) take priority over less important operations (such as memory reads).

In the current implementation, they are implemented as a pair of mutex style queues, with an additional section of logic that determines which queue a thread should join. The logic at the moment is fairly simple in that threads in the ‘write’ queue are always unlocked over those in the ‘read’ queue.

### 8.5.5 Condition Variables

The most complex synchronisation primitive is the condition variable, which has superficial similarities to mutexes. Threads can wait on a condition variable just as they can on a mutex, before another thread signals that a thread can continue.
The underlying implementation is similar to a mutex, using the same sort of thread queue system.  However, unlike a mutex, there is no inherent ownership of a condition variable - it does not necessarily guarantee that only one thread at a time can continue.

#### 8.5.5.1 Condition Variable Atomic Mutexes

However, waiting on a condition variable requires that a thread provides a locked mutex, which is atomically unlocked as part of the waiting process. When a thread is signalled to continue from the condition variable, this mutex is then atomically relocked before the thread returns. Using the same mutex between several threads allows a program to ensure thread safety, or if this is not a requirement each thread can be given a separate mutex.

The need to atomically lock and unlock mutexes mean that the code to control a condition variable is sufficiently different that has been necessary to implement it separately to the code for mutexes, despite the similarities.

#### 8.5.5.2 Condition Variable Broadcast and Signal

Condition variables allow both single threads to be unlocked one at a time, or for all the currently waiting threads to unlock at once. To do the former uses the same signal passing that is used in mutexes, whereas the latter uses the same system as barriers. Depending on the structure of the code, unblocking all the threads on a condition variable at once may simply cause them to all block on trying to acquire the mutex but this is the expected behavior.

#### 8.5.5.3 Condition Variable Contact

The typical usage of a condition variable is sleeping a thread that requires or should respond to a change in the program's state and allowing that thread to be woken when the state is changed.

However, the design of the Xcore means that often the easiest way to transfer data between threads is to use the channel system rather than the shared memory that normal programs would want to use. The standard pthreads style does not lend itself easily to using this in most cases, but by adding a non-standard function to the condition variable system - called xthreads_cond_contact() - we can provide a system that allows for threads to unblock waiting threads and receive the address of the woken thread's channel.

When calling xthreads_cond_contact(), instead of passing a standard signal, we also pass the address of the thread channel owned by the thread calling xthreads_cond_contact(). The thread at the end of the condition variable's queue is woken as normal, but after locking the mutex as normal, it then uses its own channel to send it's own channel's address back to the thread call xthreads_cond_contact().

Combining this with either custom assembly level programming or the other non-standard thread communication functions that have been added allows condition variables to act in an Xcore style, whilst maintaining similar functionality in spirit to the pthreads standard.

#### 8.5.5.4 Clocks

On standard POSIX systems, it is necessary to link a condition variable to a specific clock to allow it to time threads out if they perform a timed wait. However, on the XCore it is actually easier to give each thread waiting on the condition variable their own timers due to how the Xcore handles events. This means that this functionality has no purpose and has been left unimplemented.

# 8. Bibliography

[1] Martins, G., Stanford-jason, A., & Lacey, D. (n.d.). Benchmarking I / O response speed of microprocessors. Retrieved from

https://www.xmos.com/download/private/Benchmark-Methods-to-Analyze-Embedded-Processors-and-Sy
stems%28X7638A%29.pdf [Accessed 2/5/16]

[2] Davidson, D. (1990) A parallel processing tutorial. IEEE Antennas and Propagation Magazine, 32(2), 6–19. http://doi.org/10.1109/74.80494

[3] Mattson, T. & Wrinn, M. (2008). Parallel programming: can we PLEASE get it right this time? Proceedings of the 45th Annual Design Automation …, 7. http://doi.org/10.1145/1391469.1391474

[4] IEEE Std 1003.1-2008 (Revision of IEEE Std 1003.1-2004) Standard for Information Technology–Portable Operating System Interface (POSIX®)--Base Specifications, Issue 7

[5] Liu, T., Ji, Z., Wang, Q., & Zhu, S. (2010). Research on efficiency of signal processing on embedded multicore system. Proceedings - 2010 1st International Conference on Pervasive Computing, Signal Processing and Applications, PCSPA 2010, 907–911. http://doi.org/10.1109/PCSPA.2010.224

[6] Diaz, J., Muñoz-Caro, C., & Niño, A. (2012). A survey of parallel programming models and tools in the multi and many-core era. In IEEE Transactions on Parallel and Distributed Systems (Vol. 23, pp. 1369–1386). http://doi.org/10.1109/TPDS.2011.308

[7] Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., & Tallent, N. R. (2010). HPCTOOLKIT: Tools for performance analysis of optimized parallel programs. Concurrency Computation Practice and Experience, 22(6), 685–701. http://doi.org/10.1002/cpe

[8] Müller, M. S., de Supinski, B., & Chapman, B. (2009). Evolving OpenMP in an Age of Extreme Parallelism. 5th International Workshop on OpenMP, IWOMP 2009. http://doi.org/10.1007/978-3-642-02303-3

[9] Beazley, D. (2010). Understanding the Python GIL. PyCON Python Conference. Atlanta, Georgia, (C), 1–62. Retrieved from http://dabeaz.com/python/UnderstandingGIL.pdf [Accessed 9/5/16]

[10] Liu, X., Zhou, J., Zhang, D., Shen, Y., & Guo, M. (2010). A parallel skeleton library for embedded multicores. Proceedings of the International Conference on Parallel Processing Workshops, 65–73. http://doi.org/10.1109/ICPPW.2010.21

[11] Number, D., Technologies, M., Arques, E., & Sunnyvale, A. (2013). MIPS ® MT Principles of Operation, 1–20. Retrieved from https://imgtec.com/mips/architectures/multi-threading/ [Accessed: 2/5/2016]

[12] May, D. (2012). The Xmos Architecture and Xs1 Chips, IEEE Micro, vol 32.

[13] May, D. (2009). The XMOS XS1 Architecture. XMOS Ltd. Retrieved from https://www.xmos.com/download/private/The-XMOS-XS1-Architecture%281.0%29.pdf [Accessed: 2/5/2016]

[14] Hyde, R. (2003). XMOS Assembly Programming, Retrieved from https://www.xmos.com/download/private/Assembly-Programming-Manual%28X9432A%29.pdf [Accessed: 2/5/2016]

[15] Bull, J. M., & O'Neill, D. (2001). A microbenchmark suite for OpenMP 2.0. ACM SIGARCH Computer Architecture News, 29(5), 41–48. http://doi.org/10.1145/563647.563656

[16] Thouti, K. (2012). Comparison of OpenMP & OpenCL Parallel Processing Technologies. International Journal of Advanced Computer Science and Applications, 3(4), 56–61.

[17] Jin, H., Jespersen, D., Mehrotra, P., Biswas, R., Huang, L., & Chapman, B. (2011). High performance computing using MPI and OpenMP on multi-core parallel systems. Parallel Computing, 37(9), 562–575. http://doi.org/10.1016/j.parco.2011.02.002

[18] Pallister, J., Hollis, S., & Bennett, J. (2013). BEEBS: Open Benchmarks for Energy Measurements on Embedded Platforms. Retrieved from http://arxiv.org/abs/1308.5174 [Accessed: 2/5/2016]

[19] Munir, A., Ranka, S., & Gordon-Ross, A. (2012). High-performance energy-efficient multicore embedded computing. IEEE Transactions on Parallel and Distributed Systems, 23(4), 684–700. http://doi.org/10.1109/TPDS.2011.214

[20] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., & Brown, R. B. (2001). MiBench: A free, commercially representative embedded benchmark suite. Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538), 3–14. http://doi.org/10.1109/WWC.2001.990739

[21] Iqbal, S. M. Z., Liang, Y., & Grahn, H. (2010). ParMiBench - An open-source benchmark for embedded multiprocessor systems. IEEE Computer Architecture Letters, 9(2), 45–48. http://doi.org/10.1109/L-CA.2010.14