James Chicken

Hydra

Automatic Parallelism
Using LLVM

Computer Science Tripos, Part II

Homerton College

8th June 2014

The Lernaean **Hydra** (Λερναία Ὕδρα) was, in Greek mythology, a serpent-like water monster with many heads. For each head cut off, it grew two more.

This project aims to take serial programs and transform them into parallel ones, running across multiple threads or machines. This is similar in spirit to how the Hydra, starting life as a single-headed beast, gradually ended up with many.

The cover image was painted by 19th-century French artist Gustave Moreau, his depiction of the Lernaean Hydra. It is in the public domain.

Proforma

Name:	**James Chicken, Homerton College**
Title:	**Hydra – Automatic Parallelism Using LLVM**
Examination:	**Computer Science Tripos, Part II, June 2013**
Approx. Word Count:	**11,921**
Project Originator:	**James Chicken and Malte Schwarzkopf**
Project Supervisor:	**Malte Schwarzkopf**
Special Difficulties:	**None**

Original Aims of the Project

Design and implementation of a compiler extension which can automatically add parallelism to serial programs. This is done by offloading function calls to a parallel thread when beneficial. Static analysis is used to determine where offloading is possible, and to assess its profitability. To do this in a language and architecture-independent fashion, a modern compiler framework (LLVM) is extended to perform the analysis and transformation on its intermediate representation. Finally, a portable parallel runtime for running the parallelised programs must be implemented.

Summary of Work Completed

Despite far exceeding the anticipated complexity, the project has been very successful. My prototype, Hydra, is able to statically identify functions which are fit for offloading. It also assesses, through a variety of heuristics, the gains associated with offloading a call. It can then realise the offload intent using two supported runtimes: kernel threads and a portable thread pool implementation. The correctness of Hydra has been verified through carefully designed tests. Hydra achieves a 6.33× speedup on a serial implementation of an n-body simulation when using a 48-core machine, without any user assistance.

Declaration of Originality

I James Chicken of Homerton College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used in any substantial extent for a comparable purpose.

Signed:

Date: 8th June 2014

Contents

Chapter 1

Introduction

1.1 Motivations

Concurrent and distributed systems have become a ubiquitous part of the modern computing landscape. Processor clock speeds have plateaued in recent years, and we are no longer able to write simple serial software and expect it to run twice as fast every couple of years. The path to faster software is clear: exploit more parallelism.

This is, of course, far from a new revelation: in a modern computing system, we try to extract parallelism from our programs at many levels. *Instruction-Level Parallelism* (ILP) is the lowest of them, and tends to work rather well. However, there are clear limits to ILP, as outlined by Wall [1].

At a higher level, we may extract *Thread-Level Parallelism* by defining multiple threads of execution. This is useful if our program has naturally parallel elements – e.g., the independent physics and AI parts in a game engine. The burden is on programmers to find and exploit this kind of parallelism, which makes sense as they are best placed to identify independent, higher level concepts in their programs.

However, writing parallel code is difficult – it may create subtle bugs and security vulnerabilities which are hard to find and diagnose. Speedups are also not guaranteed, and may be undermined by threading overheads and synchronisation. Finally, this approach will not work well on legacy code.

In this project, I aim to use automatic methods to extract a finer-grained level of parallelism. It is natural for a compiler to reason at a lower level than the high-level parallelism described above – the compiler understands the dependencies of the code and the overheads of the threading model. To achieve this, the system I built, **Hydra**, parallelises function calls so that the caller may resume working before the callee has finished.

1.2 Challenges

Automatic parallelism comes with many challenges. The first problem one faces is where to start looking for parallelism. We are generally uninterested in gains of just a few instructions due to the synchronisation costs associated with threads – such gains fall under the domain of ILP. On the other hand, large gains can be very difficult to find and reason about. Can we expect our compilers to understand that components of a game are separate? As much as we can not expect the compiler to write a program for us, it seems similarly unreasonable to expect it to discover these kinds of independent high-level concepts in the program.

We need low-level constructs that our compilers can reason about, but which will give sufficient gains when parallelised. A few come to mind: loops, basic blocks and function calls. Loop parallelisation is already a well-explored area, and hence Hydra focuses on the latter two.

The next challenge is proving independence between these constructs. Independence makes parallelisation possible, whether it be independence of loop iterations, certain function calls or basic blocks. This can be quite challenging in practice, and we may respond by restricting the problem or asking the programmer for annotations. Hydra takes the former approach, and requires no programmer input whatsoever.

Finally, there is the choice of threading model. This can have quite serious implications, since Hydra must reject anything for which the gained parallelism does not balance the added synchronisation. Hydra explores two different threading models, one of which (§2.2.1) is much more restrictive than the other (§2.2.2).

1.3 Related Work

1.3.1 Cilk and OpenMP

Cilk [2] and OpenMP [3] aim to empower the programmer with concurrency primitives in C and C++. This is similar to Hydra in that they allow delegation of function calls (see §2.1) and that they provide a runtime system to efficiently implement such primitives. They differ in that Hydra aims to work without any programmer input, while Cilk and OpenMP require the source code to be annotated. In addition, Cilk and OpenMP primarily target C and C++, whilst Hydra aims to be independent of source language (see §2.4.1).

1.3.2 HELIX

HELIX [4] is a new project which aims to automatically extract loop parallelism from independent iterations of a loop. Like Hydra, it works without programmer input and harnesses LLVM to support a variety of languages and targets (see §2.4.1). As discussed

in §1.2, Hydra does not focus on loop parallelism, so Hydra and HELIX are orthogonal and complementary.

1.3.3 Vectorisation

Unlike automatic parallelisation, many modern optimising compilers feature automatic *vectorisation*, for instance, GCC [5] and LLVM through its Polly project [6]. Vectorisation is orthogonal to parallelisation and now considered a separate field, but Hydra is inspired by the impressive speedups possible, with no programmer input, from vectorisers.

1.3.4 Thread Pools

Hydra provides an implementation of a thread pool (see §3.7), but many of these already exist, for instance `libdispatch`[1] and the Windows Thread Pool.[2] These are of high quality, but are not independent of source language and target architecture; hence, a portable thread pool runtimes was implemented for Hydra.

[1] `http://bit.ly/1nwKEkd`
[2] `http://bit.ly/1jvF9zC`

Chapter 2

Preparation

In this chapter, I will explain all the work that was completed before any code was written: the underlying theory (§2.1 and §2.2), requirements identified for a successful implementation (§2.3), libraries and tools used (§2.4 and §2.6), details of some early design (§2.5) and principles of engineering established (§2.7).

2.1 Introduction to Function Offloading

In the project proposal, I used an example to illustrate what the substance of Hydra was going to consist of, repeated in Listings 2.1 and 2.2 for brevity. Listing 2.1 shows the body of a function, `f`, prior to the transformation, where `//...` represents some independent work. Listing 2.2 shows the same function after the transformation. For more details, see Appendix C.3.1.

```
void f(float x) {
  float y;
  y = sqrt(x);

  //...


  std::cout << y;
}
```

Listing 2.1: Proposal example before transformation.

```
void f(float x) {
   float y;
   std::thread t(sqrt, x, y);

   //...

   t.join();
   std::cout << y;
}
```

Listing 2.2: Proposal example after transformation.

These listings illustrate the spirit of Hydra, but raise some important questions: What is gained in the transformation? When is it worthwhile? More generally, what is it that one has managed to exploit when one makes this kind of transformation? These

are the questions which I needed to understand before I could begin designing and implementing Hydra.

2.1.1 Functions as Tasks

Abstractly, we can think of a function call as the delegation of a task. The caller would like some result to be computed, so it delegates the computation of that task to a callee. The callee then computes the required result, and returns it to the caller, which may then resume its computation.

We can apply this way of thinking to the example above: `f` needs the square root of `x`, so it delegates the task of computing it to the `sqrt` function. `sqrt` will eventually come back with the answer, which `f` then stores in `y` and continues.

This way of thinking describes the caller and callee as two separate workers. However, in a single-threaded program, we 'serialise' this setup. Control of the thread from the caller to the callee at the call site, effectively suspending the caller while the callee gets to work, illustrated in Figure 2.1.

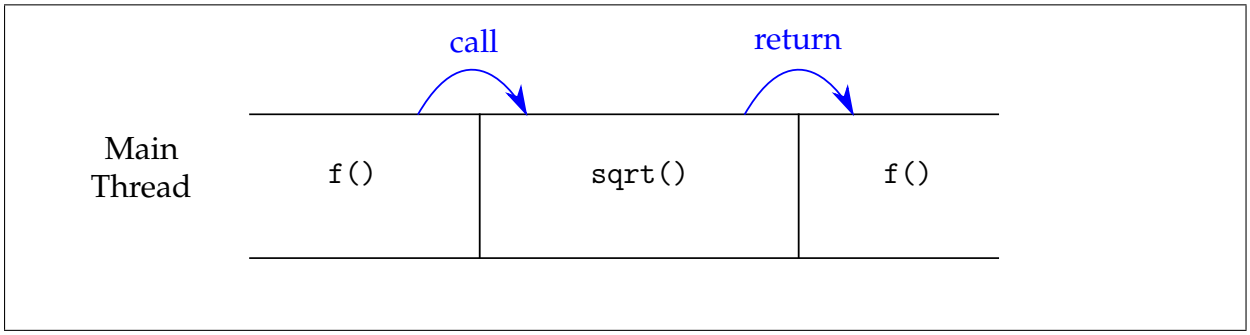

Figure 2.1: Serial Execution of Listing 2.1.

On a serial machine, this is fine, but it leaves much to be desired on today's multi-core processors, or indeed large distributed systems. If we have more resources available to us, it may make sense to run the callee in parallel.

Figure 2.2 illustrates a possible parallel schedule of Listing 2.1. There may be overheads associated with the delegation, but if `f` has enough independent work to do while `sqrt` is running, we may repay these costs and then some.[1] This leads to the condition which tells us whether the delegation is worth our while.

Denote the independent work of `f` with `sqrt` as f_{extra}, and the work required to do the delegation as $spawn$. In the serial case, the overall cost is additive:

$$SerialCost = cost(f_{extra}) + cost(sqrt)$$

[1] Figure 2.2 is optimistic in depicting synchronisation cost – an industrial strength implementation of `sqrt` is likely run faster than even idealised synchronisation times.

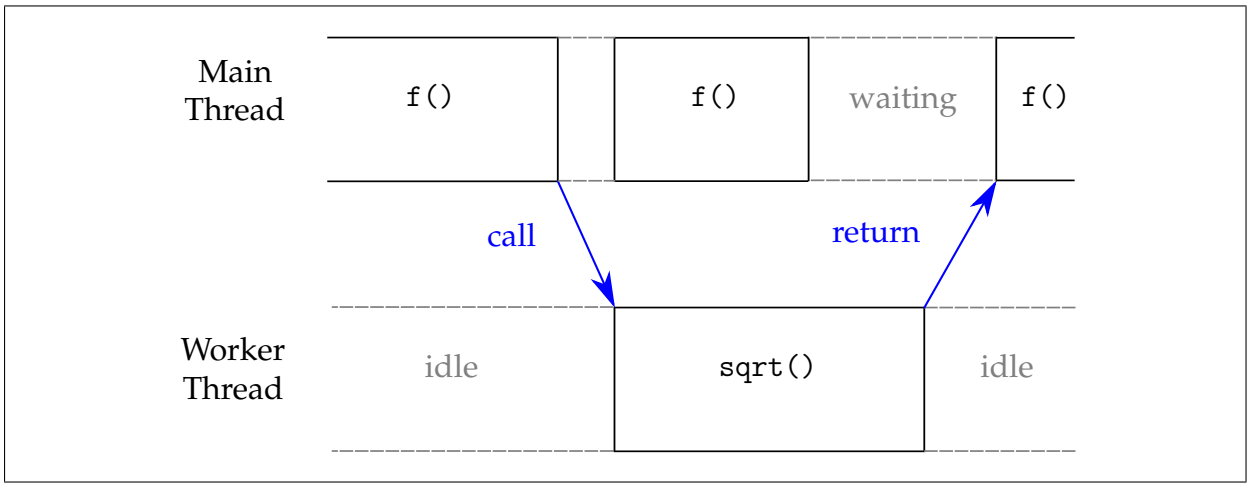

Figure 2.2: Parallel Execution of Listing 2.1.

In the parallel case, we are interested in the critical path. Thus, the overall cost is the *maximum* of the two tasks. However, must also add the overheads of synchronisation. Hence:

$$ParallelCost = max\{cost(f_{extra}), cost(sqrt)\} + cost(spawn)$$

We have made a profit if $ParallelCost < SerialCost$, leading us to *The Profitability Condition*, summarised in Figure 2.3.

$$\begin{gathered} max\{cost(caller), cost(callee)\} + cost(spawn) \\ < \\ cost(caller) + cost(callee) \end{gathered}$$

Figure 2.3: The Profitability Condition.

The Profitability Condition tells us that profitability is tied closely to $cost(spawn)$. In fact, on an ideal computer without synchronisation costs, the condition becomes:

$$max\{cost(caller), cost(callee)\} < cost(caller) + cost(callee)$$

This is *always* true for non-zero work. Thus, if we can find *anything* for the caller to do in parallel, the transformation is *always* profitable.

In practice, we will always have some synchronisation costs, but we have learnt that improvements to the synchronisation costs allow us to profitably apply this transformation more liberally and extract more parallelism from the program. It also tells us that estimating these spawning overheads is important – estimates which are too conservative are very restrictive on the kinds of gains we can make, while estimates which are too

liberal will cause slowdowns in the program due to applications of the transformation which are not profitable. This motivates further research on threading (§2.2) and the implementation of the Thread Pool (§3.7).

2.1.2 Limitations

Other than the synchronisation costs discussed in §2.1.1, there are further limitations to function offloading that are worth discussing. First, there is the problem of finding independent work that the caller can complete whilst the callee is running. In practice, this it a hard problem, since it can be difficult to know what can be done safely at compile-time.

For instance, in Listing 2.3, we may consider offloading `foo` to another thread and running `bar` locally. However, we must proceed cautiously, since if `p` and `q` alias each other, `foo` and `bar` may race and result in undefined behaviour. Whenever we apply compiler optimisations, we must prioritise correctness above all else, and therefore ban the transformation.

```
int foo(int*);
int bar(int*);

int f(int* p, int* q) {
  int x = foo(p);
  int y = bar(q); // do p and q alias?
  return x + y;
}
```

Listing 2.3: It is unclear whether `p` and `q` alias.

There are a few solutions to this. The simplest and most restrictive is to simply declare any function which takes pointer arguments as unfit for the transformation. This is the approach taken by Hydra (see §3.1), but for the sake of completeness, I will discuss other possible solutions.

- Sufficiently strong *Alias Analysis* results could permit the transformation. If we can prove that `p` and `q` do not alias, then the transformation is completely safe.
- A guarantee that `foo` and `bar` *do not modify* any memory reachable through their arguments could also be used to justify the transformation.
- Permission from the programmer to make the transformation might be given. If the programmer knows that `p` and `q` never alias, they may be able to communicate that to the compiler.[2] This can become a maintenance problem, however, since the assumption may fail to hold in future iterations of the code.

[2] E.g. by using the `restrict` keyword in C99 [10].

It is worth noting that, even though Hydra does not consider functions with pointer arguments, primitive alias analysis is still necessary. If, when analysing Listing 2.4, we cannot recognise that the program is using an alias of y, we may decide to join too late and violate the correctness of the program. This is resolved by SSA form (see §2.4.2).

```
void f(float x) {
  float y;
  float *alias = &y; // create an alias for y
  y = sqrt(x);

  //...

  std::cout << *alias; // use that alias
}
```

Listing 2.4: y is aliased before the call to `sqrt`.

Another limitation is that code is often not written with the same mental model as is presented in §2.1.1 making it harder for the compiler to expose the parallelism. For instance, Listing 2.5 has a lot of parallelism available for extraction, but the compiler will need to rearrange the code to find it.

```
void f(float x) {
  // lots of independent work...

  float y = sqrt(x);
  std::cout << y; // use y immediately

  // lots more independent work...
}
```

Listing 2.5: `f`'s body must be rearranged to expose the parallelism.

2.2 Introduction to Threading

The inequality in Figure 2.3 motivated me to minimise the synchronisation cost inherent to spawning a task. This section examines several run-time approaches to function-offloading using threads. Note that Hydra's Analyser was designed to be independent of the target runtime,[3] and while threading is the focused target of this project, it could also target a distributed system like CIEL [7].

[3] All the Analyser needs is the cost of spawning a task, so it can compute the profitability condition (Figure 2.3).

2.2.1 Kernel Threads

We will begin with a naïve implementation and discuss its limitations. The simplest way to implement a parallelism run-time system is already illustrated by Listing 2.2 – for every new task, simply create a brand new kernel thread for it, and join with the thread when the result is needed.

This implementation of task parallelism is extremely simple. I used it to facilitate testing of the compile-time portion before any run-time work was completed, and as a benchmark reference in the evaluation (see §4.4). Unfortunately, this simplicity comes at a price – the cost of creating a new kernel thread is enormous (see §4.3). This ends up heavily restricting the transformation we would like to perform.

There are other problems with this approach. The profitability condition in Figure 2.3 masks the problem of having a finite number of cores to actually execute threads. Factoring this into the Analyser is tricky – how can it know how many cores are available at a given point in the program? This is best dealt with by the runtime, yet using kernel threads ignores the problem entirely.

Finally, there is also the problem of creating a very large number of kernel threads on any operating system: *thrashing* may occur. This is where CPU utilisation is *reduced* by the presence of too many kernel threads due to pressure on the scheduler.

2.2.2 Lightweight Threads

The discussion of kernel threads above motivates us to consider alternative approaches. Hydra uses a *thread pool* of kernel threads initialised at the start of the program, and schedules tasks onto these worker threads via the pool.

The most interesting question is what happens at spawn-time. I will explore two possibilities, referred to as *eager* and *lazy* thread pools. For both, the behaviour is the same if there is a worker thread available at spawn time, shown in Figure 2.4. Note the similarity to Figure 2.2, since this is the behaviour we would expect of any task concurrency implementation when resources are available. Nonetheless, we are improving on kernel threads by paying the cost of creating threads only once at program initialisation.

I have been deliberately vague about whether the top thread in Figure 2.4 is the main thread or a worker thread. In fact, worker threads may ask the thread pool to perform tasks on their behalf just like the main thread.

Eager and lazy thread pools differ in their behaviour when all the worker threads are engaged. We shall see that the behaviour of eager thread pools is easier to implement, which is why it was ultimately used (see §3.7).

Eager Thread Pool

At spawn time, an eager thread pool will attempt to service the spawn request immediately. If all the worker threads are currently engaged, the eager thread pool informs the

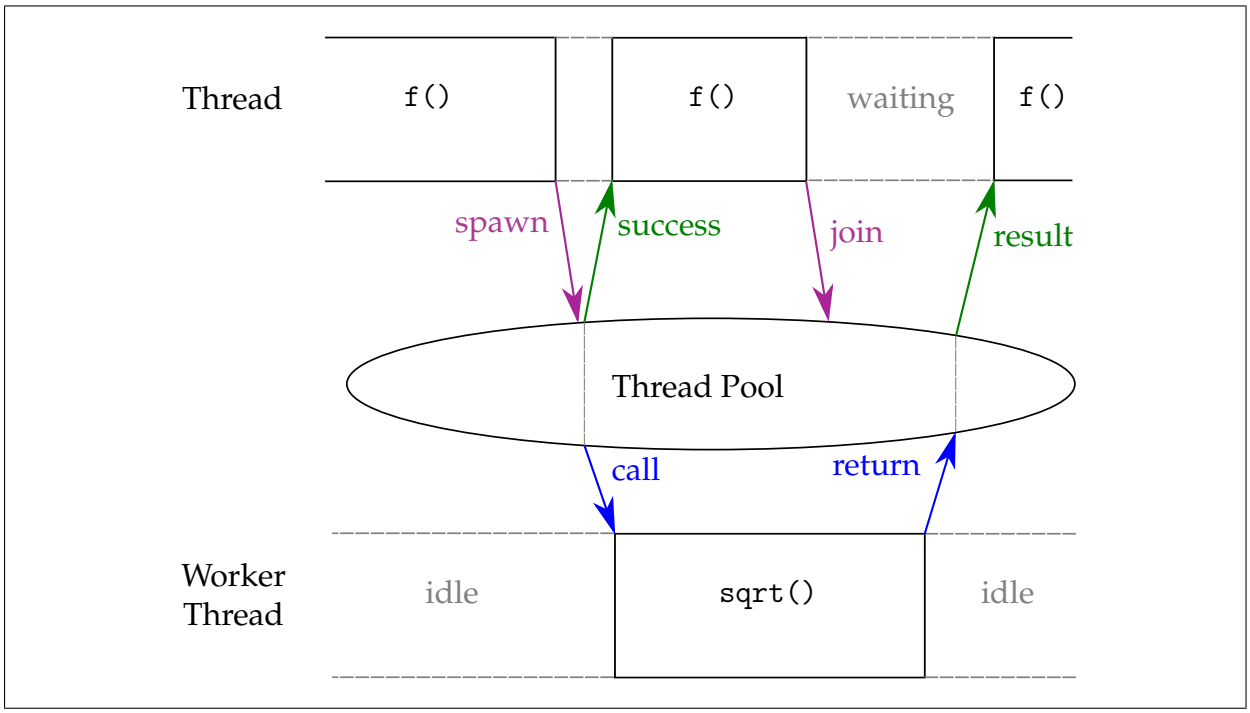

Figure 2.4: A thread pool executing Listing 2.1.

caller that their request to spawn a task failed, so the task must be run serially by the spawner, illustrated in Figure 2.5.

Figure 2.5 is similar to Figure 2.1 because we have essentially fallen back to serial behaviour. This is quite a graceful solution, but is not perfect.

First, we pay a small penalty when we fail to spawn. The penalty is likely to be small (examined in §4.3), but the existence of this cost is a regression from Figure 2.1. We must hope that these penalties do not outweigh the gains we make from successful spawns.

In addition, due to the greedy nature of an eager thread pool, we may under-extract the available parallelism in certain examples. For instance, in Figure 2.5, the eager thread pool rejected the task even though one of its worker threads became available shortly afterwards. Perhaps it would have been worth waiting a little longer so that the function could have been offloaded to this thread.

Lazy Thread Pool

By comparison, a lazy thread pool *will never fail* to spawn a task, even if no worker threads are currently available. Instead, it will add to its internal queue and wait for worker threads become available, illustrated in Figure 2.6.

This permits simplifying our spawning logic (no need to check if threads are available), which may reduce spawn time a little. However, we may wait longer for the result if the task runs much later than it was spawned. This violates the assumptions made when calculating the profitability condition (Figure 2.3) and can reduce the responsiveness of

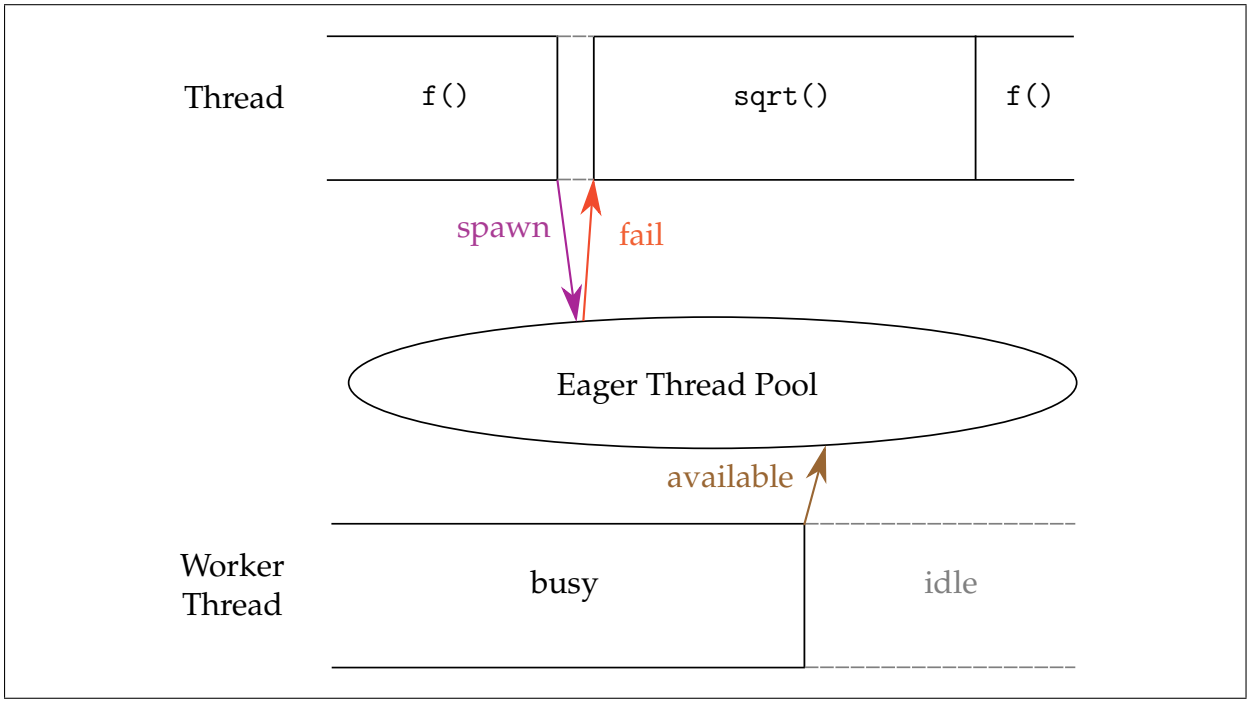

Figure 2.5: An eager thread pool executing Listing 2.1.

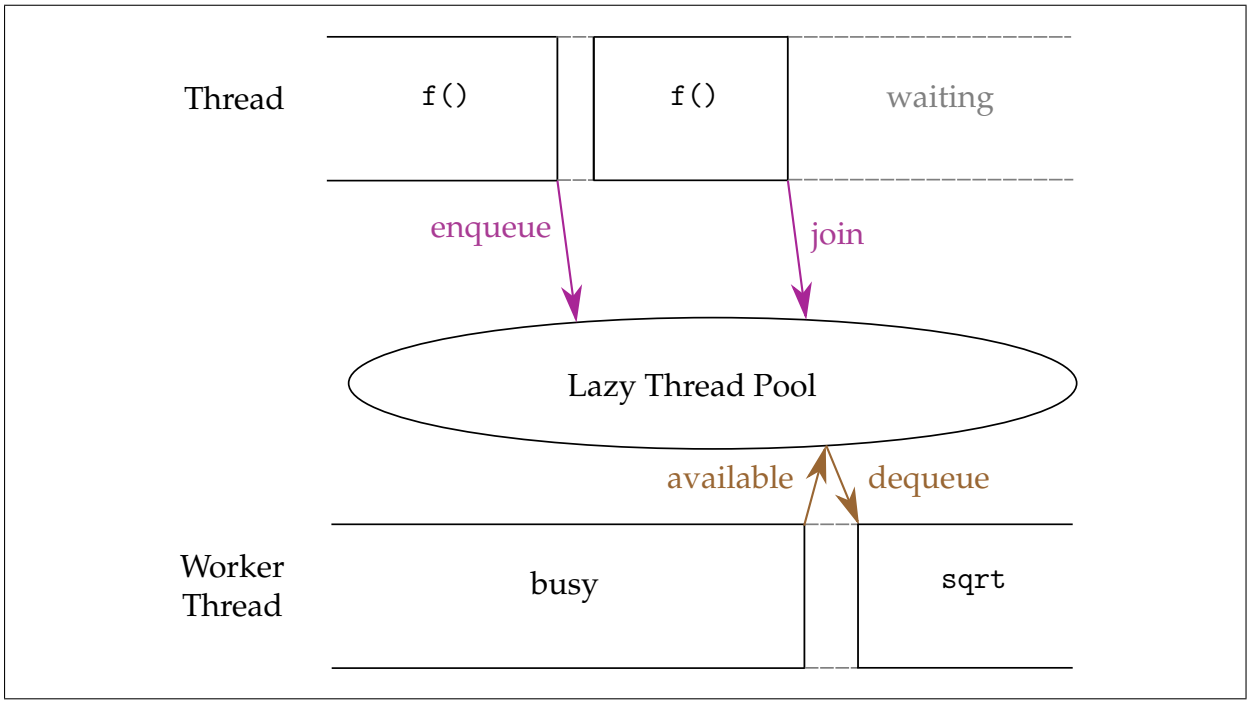

Figure 2.6: A lazy thread pool executing Listing 2.1.

the application, even though throughput may have increased. A lazy pool also carries more state than an eager one which may be a concern if memory is tight.

Alternatives

We can overcome the limitations of lazy pools with a work-conserving approach – i.e. by running the spawned work if it has not begun at *join-time*. This allows a small window for threads to become available, without running into the case where several threads are all waiting for their tasks to finish (possibly in *dead-lock*), even though those tasks are just sitting on the queue.

Another alternative is to use a threading model *without joins*. To achieve this for the program in Listing 2.1, we would need to split `f` into three parts, `f_before`, `f_during` and `f_after`, and communicate a dependency graph to the runtime. Here, `f_before` has no dependencies, `f_during` & `sqrt` depend on `f_before`, and `f_after` depends on `f_during` & `sqrt`. This approach is employed in projects like `libdispatch` (see §1.3.4).

2.3 Requirements Analysis

Hydra divides naturally into three core elements – the *Analyser*, the *Transformer* and the *Thread Pool*. This facilitates modularisation, and allows their requirements to be specified separately.

Consulting the success criterion from the project's proposal (Appendix C.4), the key requirements for my project are as follows:

Analyser — *high priority*
: Required to determine which functions are fit for spawning and, through various heuristics and applications of the profitability condition (Figure 2.3), at which callsites of those functions offloading is profitable in a supported runtime.

Transformer — *medium priority*
: Required to do all that is necessary to communicate to a supported runtime that certain function calls, which the Analyser deemed appropriate, should be offloaded.

Thread Pool — *low priority*
: Required to be an implementation of a supported runtime of the Analyser and Transformer which performs function offloading using an *Eager Thread Pool* design (§2.2.2).

In the requirements above, I used the phrase *supported runtime* to refer to a runtime system, supported by Hydra, capable of performing function offloading (§2.1).[4]

The priority of each component was important to identify. I decided that the Analyser is the most crucial part of Hydra, and so was given the highest priority. The Transformer

[4]The two supported runtimes of Hydra are *Kernel Threads* (§2.2.1) and the *Thread Pool* (§3.7).

is necessary for performing most of this project's empirical evaluation (see §4), so came next. Finally, the Thread Pool was assigned the lowest priority since it is not required for a minimum successful project. Despite this, the Thread Pool was still an important component, since the project was unlikely to offer performance gains without it.

Dependencies between these elements are small, but worth discussing. The Analyser needs to know the cost of offloading on all supported runtimes so that it can compute the profitability condition, although this can be 'stubbed out' for the purposes of development. The Thread Pool has very few dependencies on the rest of the project.

The Transformer, however, needed interfaces to the Analyser and all supported runtimes to be specified before its development could begin. Thus, specifying these interfaces was an important first step in Hydra's implementation.

2.4 The LLVM Compiler Framework

The LLVM modular compiler framework [8] is core to Hydra. I spent the vast majority of my research time learning about it and will discuss what I learnt here.

2.4.1 The Need for a Compiler Framework

When I had the idea for Hydra, I wanted to focus on the more general problem of automatic parallelism, without concentrating on issues specific to certain languages or targets. However, I did not want to create a disproportionate amount of additional work, since my vision was for Hydra to be about automatic parallelism, rather than a language and target-independent compilation.

These ideas of language- and target-independence are well known in the field of compiler construction. Rather than writing MN compilers to compile M source languages to N target architectures, an *Intermediate Representation* (IR) is established, illustrated in Figure 2.7. Armed with an IR, adding a new language involves writing one *Front-End*, and adding a new target involves writing one *Back-End*.

The use of an IR gives us a natural way to write source-target independent optimisations, by using an *IR-to-IR Optimiser*, which is sometimes referred to as the *"Middle-End"* of the compiler. This is where Hydra lives.

2.4.2 LLVM's Intermediate Representation

LLVM is centred around a well-specified intermediate representation, the *LLVM IR*. Unlike other IRs like Java Bytecode, LLVM IR is specifically designed for compiler optimisations, rather than interpretation. As such, LLVM IR features *Three-Address Instructions* and has *Static Single Assignment* Form. It is also organised into *Basic Blocks*. All of these simplify the task of finding join points (see §3.3).

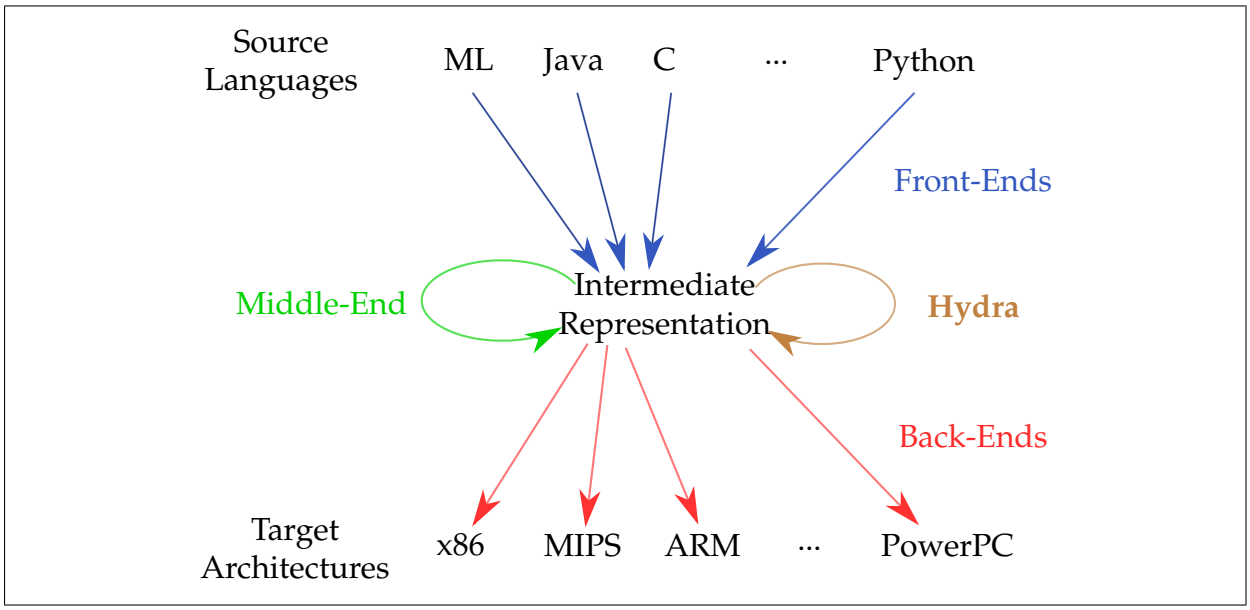

Figure 2.7: The Compiler Hourglass.

Three-Address Instructions

Three-Address Instructions are a feature of a register-register architecture. Each instruction features up to two registers as an argument, and one register for the result and there are no implied arguments. This is in contrast to JVM instructions, which implicitly use the *argument-stack* for arguments and results. Three-address instructions can make code larger, but have the advantage of making dependencies clearer.

The differences between stack and three-address instructions are illustrated by Listings 2.6, 2.7, and 2.8.

```
a = b - c;
b = c - d;
```

Listing 2.6: High-Level Language Extract.

```
push b   ; stack = { b }
push c   ; stack = { b, c }
sub      ; stack = { (b-c) }
pop  a   ; stack = {}

push c   ; stack = { c }
push d   ; stack = { c, d }
sub      ; stack = { (c-d) }
pop  b   ; stack = {}
```

Listing 2.7: Stack Code for Listing 2.6.

```
load  R1 b       ; R1 := b
load  R2 c       ; R2 := c
sub   R3 R1 R2   ; R3 := R1-R2
store a  R3      ; a := R3

load  R4 c       ; R4 := c
load  R5 d       ; R5 := d
sub   R6 R4 R5   ; R6 := R4-R5
store b  R6      ; b := R6
```

Listing 2.8: Three-Address Code for Listing 2.6.

Static Single Assignment (SSA) Form

When we perform optimisations, we must respect the dependencies of the instructions in the program. We can classify dependencies into three groups: Read-after-Write, Write-after-Read and Write-after-Write, illustrated in Figure 2.8.

Read-after-Write:

```
load  R1 x
; ...
add   R2 R1 R1
```

Write-after-Read:

```
store x  R1
; ...
load  R1 y
```

Write-after-Write:

```
load  R1 x
; ...
load  R1 y
```

Figure 2.8: Instruction Dependencies.

Read-after-Write dependencies are known as 'true dependencies' which must be respected. The other two 'dependencies', however, are caused by instructions corrupting each other's registers. If there was another free register, this corruption could be avoided by writing into that free register instead. As such, these last two kinds are often known as 'false dependencies'.

Code which is in SSA form has *no false dependencies*, because each register is statically assigned once. With an unbounded number of registers, every program has an SSA form, and valid LLVM IR is required to have this form.

Basic Blocks

A basic block is a sequence of instructions which have no interesting control-flow. As such, all instructions within a basic block are *mutually executed* – i.e. if one of the instructions is executed, then all of them are. In LLVM IR, the final instruction of a basic block is known as the terminator instruction.

A function in LLVM IR can be represented as a directed graph of basic blocks linked together by their terminator instructions. Such a graph is known a 'Control-Flow Graph' (CFG). Listing 2.9 and Figure 2.9 shows a C program and its representation as a CFG.

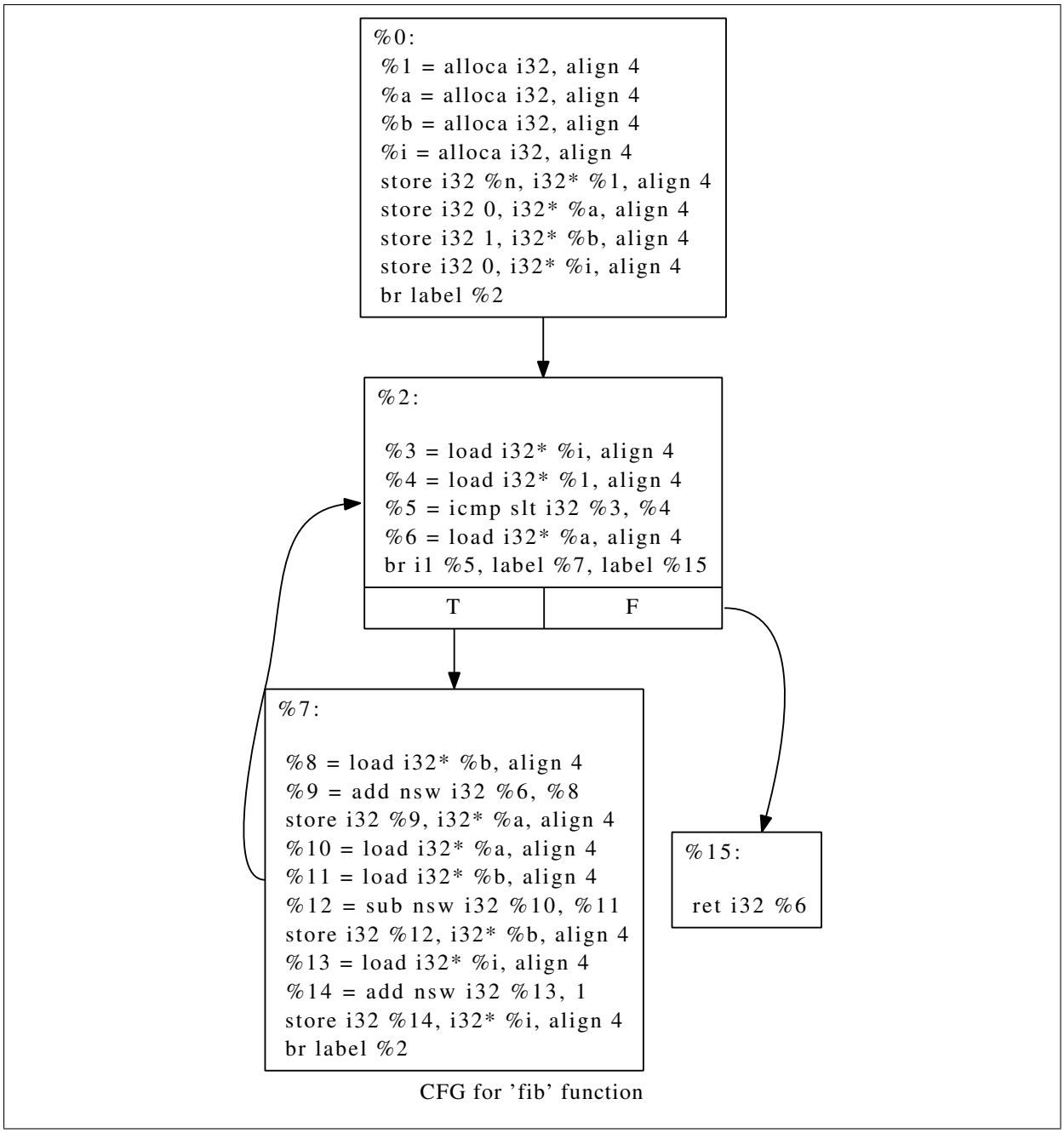

Figure 2.9: Control-Flow Graph corresponding to Listing 2.9.

```
int fib(int n) {
  int a = 0;
  int b = 1;
  for (int i = 0; i < n; ++i) {
    a = a + b;
    b = a - b;
  }
  return a;
}
```

Listing 2.9: C program for the n^{th} Fibonacci number.

2.4.3 LLVM Optimisation Passes

Hydra's Analyser and Transformer are made up of LLVM optimisation passes. These passes can be dynamically loaded by LLVM's target independent optimiser.

When designing an LLVM optimisation pass, the visibility of the pass is the first choice one must make. I selected the `ModulePass` for Hydra's passes, since these allow for the greatest visibility, though possibly at the expense of poorer cache behaviour.

2.4.4 Useful Existing LLVM Analysis Passes

LLVM provides several built-in analysis passes. Hydra's Analyser uses a couple of these to improve the quality of function cost estimation (see §3.2) which are discussed below.

Call Graph Construction is used to build a *Call Graph* – a directed graph where nodes represent functions and the edges represent one function calling another.

Loop Analysis exists to find and analyse loop structures in LLVM IR. Amongst them, *Scalar Evolution* can attempt to find a loop's trip count.

2.5 Implementation Approach

In §2.3, I stated that Hydra's Analyser and Transformer (collectively the 'compile-time' component) were assigned the highest priority, so it was important to describe their design early. I considered using two passes – one for the Analyser, and one for the Transformer. However, these two passes would be quite monolithic, and inhibitive of modularisation. Hence, I ended up designing four analysis passes and two transformation passes, given below. Their dependencies are illustrated by Figure 2.10.

Fitness exists to filter out functions which are not fit for offloading. As discussed at the end of §2.1.1, we are not interested with functions which have pointer arguments or modify global variables.

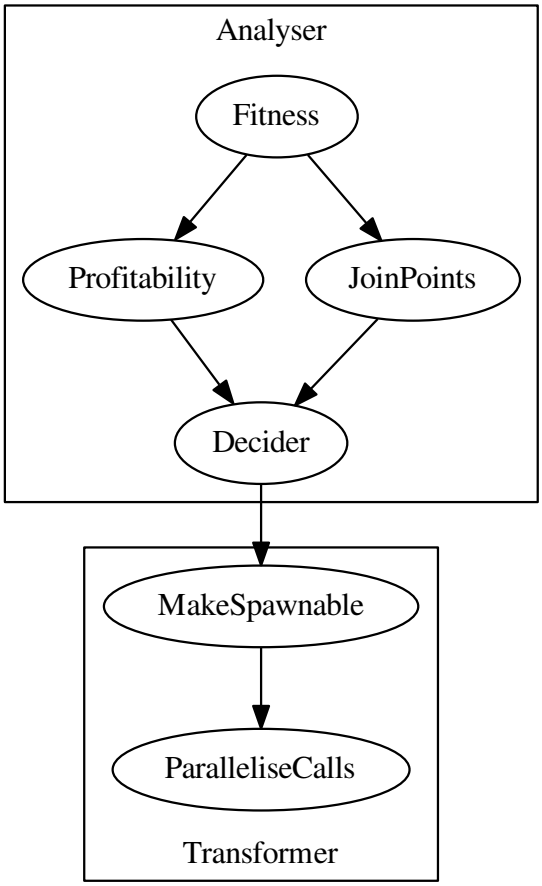

Figure 2.10: The architecture of Hydra's compile-time portion.

Profitability tries to estimate the cost of calling each function. This essentially calculates the $cost(callee)$ from the Profitability Condition (Figure 2.3).

JoinPoints exists to find where to join with the callee would be. Pushing this back as far as possible will yield better results, but the correctness of the program cannot be altered by violating instruction dependencies.

The Decider gets the final say on whether we shall parallelise. It can see all the data gathered from previous analysis passes. The decision is made by applying the Profitability Condition (Figure 2.3), but to do this, it must calculate $cost(caller)$.

MakeSpawnable synthesises a 'spawnable' version of each function which the Decider chose to spawn. This helps Hydra deal with return values and user defined types.

ParalleliseCalls does the 'dirty work' of the Transformer, by replacing each call which the Decider chose to parallelise with spawn intrinsics for the target runtime, and switching all the old users of the return value with the new return value.

2.6 Choice of Tools

2.6.1 Programming Language — C++

C++ is the most natural implementation language for Hydra. The primary reason is that the interface to LLVM is a C++ interface. In addition, I was already proficient in C++ when proposing the project. I decided to use C++11 for Hydra, as discussed in Appendix A.

2.6.2 Libraries

The libraries used by Hydra are listed in Table 2.1. Of them, the library of key importance is the *Threading Library*, for which I turned to C++11 for its simple and portable interface [9].

2.6.3 Development Environment

Development tools used are summarised in Table 2.2.

The rationale for using Clang as the primary compiler was twofold. First, it has excellent support for C++11, and is able to emit LLVM IR, which was extremely useful for testing and evaluating the project (see §4). In addition, the sanitisers of Clang, particularly *ThreadSanitizer* [sic] were excellent for debugging.

Revision control and backup were also of key importance. I used the online service *BitBucket* to host my project's repository, as well as regularly pulling backups onto my

[5]The implementer chooses the license. libstdc++ and libc++, are GPLv3 and MIT licensed, respectively.

Library	*Version*	*Purpose*	*License*
LLVM	3.4	*LLVM IR* representation and manipulation, *LLVM Optimisation Pass* infrastructure	University of Illinois
C++ Standard Library	C++11	*Threading*, Algorithms, Containers, Numerics, Random-Number Generation	Various[5]
Boost	1.53	Graph data structure and algorithms	Boost Software License

Table 2.1: Libraries used by Hydra.

Tool	*Version*	*Purpose*	*License*
Ubuntu	13.10	Operating System	GPL
Clang	3.4	Compiler	UIUC License
Vim	7.4	Text Editor	Vim License
Autoconf	2.69	Build Configuration	GPLv3
Make	3.81	Build Tool	GPLv3
GDB	7.6.1	Debugger	GPLv3
Git	1.8.3	Revision Control	GPLv2

Table 2.2: All tools used in the development of Hydra.

two development machines. I also used this repository with my LaTeX source files for the proposal and dissertation.

2.7 Software Engineering Techniques

I used an *Iterative Development Model* when engineering the project. Proper modularisation and encapsulation of certain parts of the software was vital to allow each of the components to be developed in isolation, and I used C++ features of classes, namespaces, access control and immutability to aid in this. In addition, proper interfaces were specified using class declarations, and header files were carefully designed to be self-contained and include only the files needed.

In addition, I maintained a level of discipline when developing the software and adhered to good practices, such as treating compiler warnings as errors, adding comments where the code is subtle or unclear, programming in a structured and defensive style, and using assertions where appropriate.

2.8 Summary

In this chapter, I have discussed the work completed before the implementation began. I discussed the surrounding theory in detail, and talked about the design goals of Hydra, as well as the tools and techniques used – in particular, how LLVM is fundamental to Hydra's success.

The next chapter will give details on how the goals and designs laid out here were achieved.

Chapter 3

Implementation

This chapter explains how the designs from the previous chapter were realised. Hydra is discussed as seven pieces: fitness (§3.1) and profitability (§3.2) of functions, finding join points (§3.3), the final decision (§3.4), making functions spawnable (§3.5), parallelising calls (§3.6) and the Thread Pool runtime (§3.7).

3.1 The Fitness Analysis Pass

The first thing done in the Analyser is to check which functions of the program can possibly be offloaded. Doing this first makes a lot of sense, since we need not waste time performing further analysis on functions which cannot be offloaded.

3.1.1 Fitness Requirements

To be fit, a function may not have pointer arguments or access global variables. As such, these functions are 'pure' in the sense that they depend only on their arguments and never on global state.

These restrictions make the project more tractable than it would be otherwise, as discussed in §2.1.2. Any relaxations to the above must be accompanied by alias analysis, which is a hard problem. The potential gains from lifting such restrictions depends strongly on the quality of the alias analysis.

3.1.2 Pointer Arguments

To discover whether a function uses pointer arguments, Hydra looks at the function's signature. The implementation, given in Listing 3.1, checks if any of the function's arguments is of pointer type. Hydra identifies all functions with pointer arguments by running the checker on each one.

```
bool hasPointerArgs(const llvm::Function &F) {
  return std::any_of(F.arg_begin(), F.arg_end(),
    [](const llvm::Argument &arg) {
      return arg.getType()->isPointerTy();
    });
}
```

Listing 3.1: The pointer arguments checker used in Hydra.

3.1.3 Global Variables

Global variables can be identified by scanning all instructions in a function, looking for instructions with a global operand. The implementation is outlined in Listing 3.2.

```
bool referencesGlobalVariables(const llvm::Function &F) {
  return std::any_of(inst_begin(F), inst_end(F),
    [](const llvm::Instruction &I) {
      return std::any_of(I.op_begin(), I.op_end(),
        [](const llvm::Use &U) {
          return isa<GlobalVariable>(U) ||
                 isa<GlobalAlias>(U);
        });
    });
}
```

Listing 3.2: Check if a function reference a global variable.

A potential cause for concern here is the nested a call to `any_of` inside the predicate of another `any_of`, which may suggest that the function runs in *quadratic time*. In fact, if $insts(F)$ is the set of instructions in a function F, and $ops(I)$ is the set of operands in instruction I, then the complexity of Listing 3.2 has an upper bound of:

$$\mathcal{O}\left(\max_{I \in insts(F)} \left\{ |ops(I)||insts(F)| \right\}\right)$$

This may seem alarming, but most instructions will not have more than three operands, since LLVM uses *Three-Address Instructions* (§2.4.2). Hence, in practice, we can view the complexity as linear in the number of instructions in F, since $|ops(I)|$ is approximately a constant factor of time.

Listing 3.2 is not sufficient to catch all functions which depend on or modify global variables, because they may call a function which does this on their behalf. In Listings 3.3 and 3.4, we see a C program, and its representation in IR. `modify` directly modifies a global, and `impure` modifies it indirectly via `modify`. Listing 3.2 will return *true* for `modify` but *false* for `impure`, since none of its instructions reference a global.

```
int global = 0;



void modify() {
  ++global;
}



void impure() {
  modify();
}
```

Listing 3.3: Impure C Functions.

```
@global = global i32 0

define void @modify() {
  %1 = load i32* @global
  %2 = add nsw i32 %1, 1
  store i32 %2, i32* @global
  ret void
}

define void @impure() {
  call void @modify()
  ret void
}
```

Listing 3.4: Impure IR Functions.

3.1.4 Transitivity of Fitness

Algorithm 3.1 addresses the above issue by recursively exploring function calls as they are encountered. This, however, does unnecessary recomputation – for instance, if some function, `f`, contains three calls to g, Algorithm 3.1 will compute whether g depends on globals three times when executing on `f`.

Hydra solves this with the iterative Algorithm 3.2. This begins by marking all functions which *explicitly* reference globals, using Listing 3.2. Each iteration, it marks all functions which call marked functions, until no changes were made. The algorithm is correct because *fitness is transitive* with respect to the 'calls' relation – a fit function may only call fit functions.[1]

Algorithm 3.1 Compute whether a function is impure using recursion.

```
1: function NaiveGlobal(F)
2:     for all I ∈ insts(F) do
3:         if I is a function call then
4:             return NaiveGlobal(calledFun(I))
5:         else if I references a global then
6:             return true
7:         end if
8:     end for
9:     return false
10: end function
```

[1] Technically, this isn't 'transitivity' in the traditional sense, since we have two relations – the unary '*isFit*', and the binary '*calls*'.

Algorithm 3.2 Iteratively find all impure functions.

```
1:  CallGraph ← generateCallGraph()
2:                                                          ▷ Initialisation
3:  for all F ∈ CallGraph.functions() do
4:      if REFERENCESGLOBALVARIABLES(F) then mark F
5:      end if
6:  end for
7:                                                          ▷ Iteration
8:  repeat
9:      for all F ∈ CallGraph.functions() do
10:         if F is not marked then
11:             for all G ∈ CallGraph.calledFunctions(F) do
12:                 if G is marked then mark F
13:                 end if
14:             end for
15:         end if
16:     end for
17: until No changes were made
```

An example of Algorithm 3.2 executing is given in figure 3.1. We see a simple call graph with five functions. Only one of them, `h`, explicitly references global variables. However, after two iterations, the algorithm converges and we see that only `f` does not depend on globals.

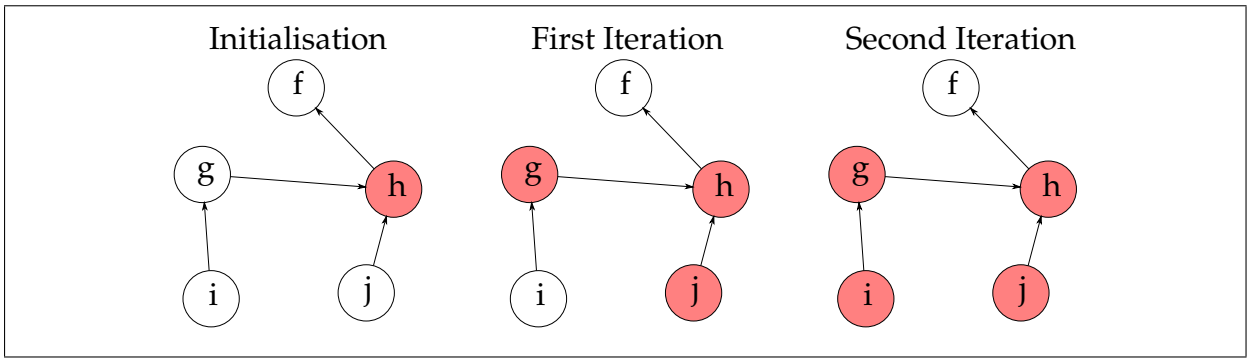

Figure 3.1: Algorithm 3.2 on a simple call graph.

3.2 The Profitability Analysis Pass

The Profitability Condition (Figure 2.3) needs to know how much work is performed by the callee – $cost(callee)$. Hydra's Profitability analysis pass is dedicated to estimating this cost for all functions which can be offloaded.

3.2.1 Difficulty of Estimating Profitability

Estimating the cost of a function at compile-time is an inherently difficult task – doing it 'properly' is undecidable, due to the Halting Problem. This seems demoralising, but since the cost of inaccuracy is that the program *might* run more slowly, we can safely use estimates.

There are several design decisions that need to be established. For a paralleliser which *never* makes the program slower, principles of one-sided error must be employed judiciously. Such a paralleliser is likely to have a high false negative rate, and commonly miss profitable parallelisation opportunities. Hydra is not such a paralleliser.

In addition, a balance must be found between good algorithmic complexity and high quality of estimation. While performance of Hydra's Analyser is not a specific requirement (§2.3), I decided to aim for asymptotic complexity of *quadratic* or less. Anything worse would have limited the tests I could reasonably perform in §4.

A goal for Hydra is target independence, but this has the consequence that the heuristics are of inherently lower quality than ones which have knowledge of a specific platform. I tried to be optimistic about capabilities of the target so that false positives are minimised. For example, it is assumed that branches never cause a stall.

3.2.2 The First Heuristic

The initial heuristic employed in Hydra was deliberately simplistic to allow for rapid end-to-end testing. I settled on the *number of instructions* in the function as the profitability estimator in the first heuristic, h_0. An implementation of h_0 is given in Listing 3.5.

$$h_0(F) = |insts(F)|$$

```
unsigned numInstructions(const llvm::Function &F) {
  return std::distance(inst_begin(F), inst_end(F));
}
```

Listing 3.5: Implementation of h_0.

An immediate refinement opportunity comes from the observation that some IR instructions are never emitted into the target, and can hence be ignored. These refinements give us heuristic h_1, implemented by Listing 3.6.

$$h_1(F) = \sum_{I \in insts(F)} cost_1(I), \quad \text{where}$$

$$cost_1(I) = \begin{cases} 1 & \text{if } I \text{ emits,} \\ 0 & \text{otherwise.} \end{cases}$$

```
unsigned numEmittingInsts(const llvm::Function &F) {
  return std::count_if(inst_begin(F), inst_end(F),
    [](const llvm::Instruction &I) {
      return !(isa<BitCastInst>(I) || isa<PHINode>(I));
    });
}
```

Listing 3.6: Implementation of h_1.

While h_1 avoids some pathological cases of h_0, they have similar characteristics. First, they both run in $\mathcal{O}(|insts(F)|)$ time, which satisfies our complexity requirements. However, they both *underestimate* the cost of *function calls* and *loops* and *overestimate* the cost of *conditional branches*. The former two are addressed in §3.2.3 and §3.2.4, while the latter is discussed in §3.2.5.

3.2.3 Dealing with Function Calls

Refining h_1 to more accurately account for the cost of function calls, we can use a recursive cost function, yielding heuristic h_2'.

$$h_2'(F) = \sum_{I \in insts(F)} cost_2'(I), \quad \text{where}$$

$$cost_2'(I) = \begin{cases} h_2'(calledFun(I)) + 1 & \text{if } I \text{ is a call,} \\ 1 & \text{if } I \text{ emits,} \\ 0 & \text{otherwise.} \end{cases}$$

There is a 'bug' in this definition, as it becomes circular if F is recursive. We can fix this by using the previous behaviour if F and $calledFun(I)$ are *mutually recursive*, yielding heuristic h_2.

$$h_2(F) = \sum_{I \in insts(F)} cost_2(I), \quad \text{where}$$

$$cost_2(I) = \begin{cases} h_2(calledFun(I)) + 1 & \text{if } I \text{ is a non-recursive call,} \\ 1 & \text{if } I \text{ emits,} \\ 0 & \text{otherwise.} \end{cases}$$

Strongly Connected Components

The naïve recursive algorithm h_2 will perform a lot of recomputation. Computing h_2 for all functions can be done *bottom-up* by walking the call graph in topological order. However, if the call graph is not a DAG, there will be ambiguous cases. The solution is

to decompose the graph into its *Strongly Connected Components* (SCCs) and process each SCC in isolation.

In a directed graph, we write $B \rightsquigarrow A$ if there is a path from B to A. If $A \rightsquigarrow B \wedge B \rightsquigarrow A$, we write $A \leftrightsquigarrow B$. '$\leftrightsquigarrow$' is an *Equivalence Relation*[2] and a graph's SSCs are defined as the equivalence classes of $\leftrightsquigarrow$, i.e. the subgraphs for which all nodes are reachable from all others. Figure 3.2 shows a graph decomposed into its SCCs.

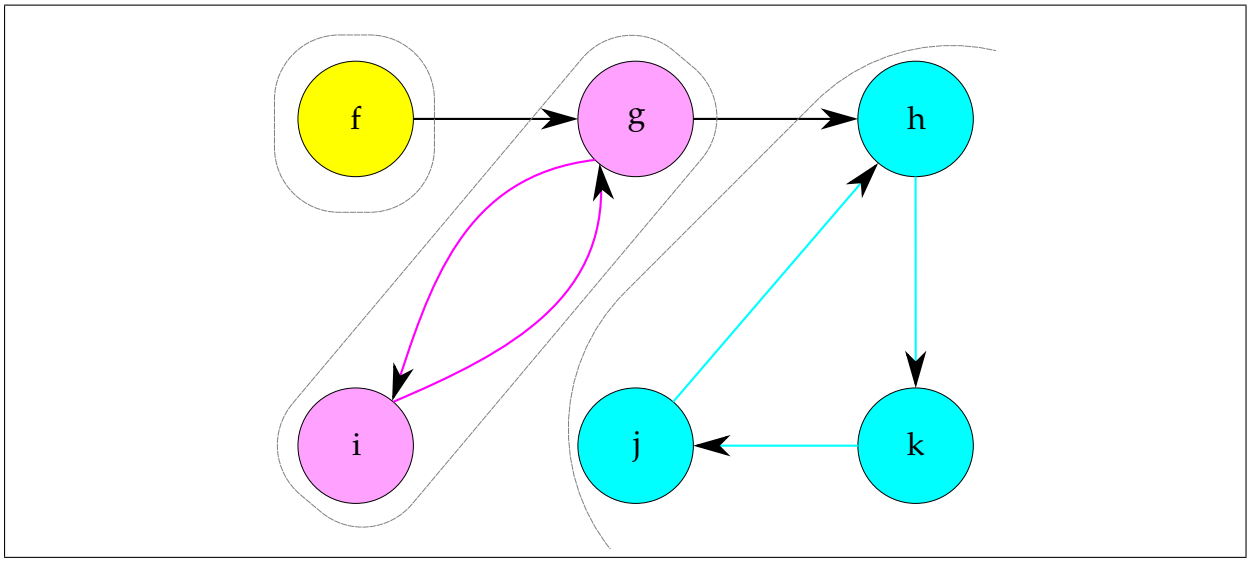

Figure 3.2: A call graph decomposed into its Strongly Connected Components.

Algorithm 3.3 computes h_2 by walking the call graph in SCC order. The assertion on line 7 holds because the bottom-up SCC traversal guarantees that the algorithm has already seen any function not in the current SCC.

Recursive Functions

h_2 falls back to the behaviour of h_1 for recursive function calls, and hence is still underestimating their cost. One improvement employed in Hydra is to compute h_2 for each function in the SCC while maintaining counts of intra-SCC calls, and then add these costs back in a post-processing step. This effectively assumes each recursive call is executed exactly once, so it is still likely to be an underestimation.

3.2.4 Dealing with Loops

All the heuristics so far underestimate the cost of loops. Listing 3.7, while contrived, is a sufficient motivator for trying to do a better job. The IR representation is nine instructions long, so $h_2(\texttt{do\_work}) = 9$. However, this code will of course run for much more instructions in practice.

[2] $\leftrightsquigarrow$ is reflexive and symmetric by definition. For transitivity, note that $\rightsquigarrow$ is transitive since if there is a

Algorithm 3.3 An algorithm to compute h_2 for all functions in the call graph.

```
1: CallGraph ← generateCallGraph()
2: cachedFuns ← generateDictionary()
3: for all SCC ∈ CallGraph.SCCsBottomUp() do
4:     for all F ∈ SCC.functions() do
5:         for all I ∈ F.instructions do
6:             if I is a call and I ∉ SCC then
7:                 assert calledFun(I) ∈ cachedFuns
8:                 instCost ← cachedFuns[calledFun(I)] + 1
9:             else if I emits then
10:                instCost ← 1
11:            else
12:                instCost ← 0
13:            end if
14:            cachedFuns[F] ← cachedFuns[F] + instCost
15:        end for
16:    end for
17: end for
```

```
int do_work(int x) {
  int ret = 0;
  for (int i = 0; i < 1000; ++i)
    ret += (i*x);
  return ret;
}
```

Listing 3.7: Code in a tight loop that will cause h_2 to underestimate the cost.

LLVM's Scalar Evolution pass (§2.4.4) can try to find the trip count for these loops. Heuristic h_3 uses this information, if it is available.

$$h_3(F) = \sum_{I \in insts(F)} tripCount(I) cost_3(I), \quad \text{where}$$

$$cost_3(I) = \begin{cases} h_3(calledFun(I)) + 1 & \text{if } I \text{ is a non-recursive call,} \\ 1 & \text{if } I \text{ emits,} \\ 0 & \text{otherwise.} \end{cases}$$

$$tripCount(I) = \begin{cases} t & \text{if } I \text{ is in a loop with trip count provably } t, \\ 1 & \text{otherwise.} \end{cases}$$

h_3 can be computed with a modified version of Algorithm 3.3. We simply add the following between lines 13 and 14:

$instCost \leftarrow instCost \times TripCount(I)$

The algorithmic complexity now depends on the complexity of ScalarEvolution, which is not documented. This suggests it is likely be linear, or perhaps quasi-linear, since LLVM has a policy to avoid quadratic complexity, except for well-documented exceptions (e.g. Alias Analysis).

3.2.5 Further Work

Due to time constraints inherent in a Part II project, I had limited scope to extend the Profitability analysis. Of course, the difficulty of the problem suggests that there will always be some restrictions, but I have some ideas of how the current heuristics might be improved in the future.

All heuristics discussed in this chapter *overestimate* the cost of *conditional branches*, since they assume all paths are taken. A couple of solutions are detailed below. However, it seems unlikely that these will have much impact to the overall result, since $cost(caller)$ is likely to the be the dominating factor of the Profitability Condition (Figure 2.3), not $cost(callee)$.

- One could build a weight graph based on a CFG and find the shortest paths. This is done in the Decider when calculating $cost(caller)$ (see §3.4.3). However, this does not interact well with the loop extensions in h_3, is quite expensive to do on every function and is very pessimistic.[3]
- Use *Profile-Guided Optimisation* (PGO) to determine the 'hot-path' and use the cost of that path the cost of the function. See §5.3 for more discussion of PGO.

path from A to B, and from B to C, then we can find a path from A to C by going via B.

[3]Imagine a function with a large loop, but a cold early-exit path. The shortest path is *much* shorter than the typical path.

As mentioned in §3.2.1, Hydra's heuristics are weaker than ones which are *target-aware*. Using a 'swappable' version would allow target-aware heuristics to be used where available, whilst maintaining the general heuristic for unknown targets.

3.3 The JoinPoints Analysis Pass

Finding where to join with an offloaded function is a mission-critical component of Hydra – getting this wrong results in correctness being is almost certainly lost. At the same time, we have a desire to place the join as late as is safely possible, to allow more parallelism to be extracted. Hydra distinguishes between *exactly-once* (§3.3.2) and *at-least-once* (§3.3.3) joining. We will see that the latter allows us to achieve better results than the former.

3.3.1 Detecting Dependencies

To find the join point, we are looking for instructions which depend on the return value of the off-loaded call. We can scan a basic block for uses of this value, which will determine if we need to join the current block, and if so, where. The implementation is given in Listing 3.8.

```
llvm::Instruction *findJoinPoint(llvm::CallInst *ci,
                                 const bb_iter I,
                                 const bb_iter E) {
  auto join = std::find_if(I, E,
    [&](llvm::Instruction &inst) {
      return std::any_of(inst.value_op_begin(),
                         inst.value_op_end(),
                         [&](Value *v) { return v == ci; });
    });
  return (join != E ? &*join : nullptr);
}
```

Listing 3.8: Implementation of Hydra's dependency finder.

Thanks to SSA form (§2.4.2), it is sufficient to compare against the return value directly without worrying about aliases. Any alias of the return value must be created after the call instruction, so will be found by the code in Listing 3.8. A more advanced implementation could attempt to track the creation of these aliases, or plug into an alias analysis.

3.3.2 Exactly-Once Joining

Joining with a C++ thread multiple times will immediately halt the program. As such, when working with the Kernel Threads runtime (§2.2.1), we need to guarantee that only one join is executed on all possible paths.

An effective way of ensuring this is by joining in the same basic block as the spawn. Instructions in a basic block have the mutual execution property (see §2.4.2) and so the exactly-once property holds trivially. Algorithm 3.4 uses Listing 3.8 to see if there are dependencies in the spawn block, and if not, places the join at the end of the block.

Algorithm 3.4 Find the exactly-once join point.

```
1: function EXACTLYONCE(call)
2:     callBlock ← call.getBlock()
3:     join ← FINDJOINPOINT(call, call, callBlock.end())
4:     if join ≠ nullptr then
5:         return join
6:     else
7:         return callBlock.getTerminator()
8:     end if
9: end function
```

3.3.3 At-Least-Once Joining

Unlike Kernel Threads runtime, the Thread Pool runtime (§3.7) can cope with joins being executed more than once. We could use Algorithm 3.4 to find joins for the Thread Pool, but with the weaker requirements, we can get better results.

Algorithm 3.5 crosses the basic block boundary by performing a *breadth-first search* of the CFG,[4] using Listing 3.8 on each block to find dependencies on the return value. At least one join must occur on *all paths*, so if we do not join in a block, we must consider all of its successors.

The algorithm has two important special cases to consider. First, if it finds a block which does not depend on the return value, but has no successors, this means there is a path through the function which does not use the return value at all. To comply with at-least-once semantics, we must emit a join at the end of such blocks.[5] Second, if control loops back around to the spawn block, we actually do not require a join before the next spawn. However, we should check for any dependencies just before the spawn itself, which is why line 20 of Algorithm 3.5 uses the range $[begin, spawn)$, rather than the entire block.

[4]The choice of breadth-first is arbitrary – using a stack instead of a queue would make the search depth-first.

[5]In these cases, it may make more sense to emit a 'detach' to indicate that the return value is no longer needed, but this is not currently supported by the Thread Pool runtime.

Algorithm 3.5 Breadth-first search to find the at least once join points.

```
1: function AtLeastOnce(call)
2:     callBlock ← call.getParent()
3:                                                  ▷ Check for join in spawn block
4:     join ← findJoinPoint(call, (call, callBlock.end()))
5:     if join ≠ nullptr then
6:         return join
7:     else if callBlock.successors() = ∅ then
8:         return callBlock.getTerminator()
9:     end if
10:                                                 ▷ Initialise the BFS
11:    joinPoints ← ∅
12:    exploredBlocks ← ∅
13:    toExplore ← emptyQueue()
14:    toExplore.enqueueAll(callBlock.successors())
15:                                                 ▷ Do the BFS
16:    while toExplore is not empty do
17:        currBlock ← toExplore.dequeue()
18:        if currBlock ∉ exploredBlocks then
19:            if currBlock = spawnBlock then
20:                join ← findJoinPoint(call, (currBlock.begin(), call))
21:            else
22:                join ← findJoinPoint(call, currBlock.range())
23:            end if
24:            if join ≠ nullptr then
25:                joinPoints ← joinPoints ∪ {join}
26:            else if currBlock.successors() ≠ ∅ then
27:                toExplore.enqueueAll(currBlock.successors())
28:            else
29:                joinPoints ← joinPoints ∪ {currBlock.terminator()}
30:            end if
31:            exploredBlocks ← exploredBlocks ∪ {currBlock}
32:        end if
33:    end while
34:    return joinPoints
35: end function
```

An illustration of Algorithm 3.5 running on a basic CFG is given in Figure 3.3. The spawn is in block 'entry', and the result is used in three blocks, A, C and D, each highlighted with a '*'. The algorithm converges in three steps. We see that, even though D uses the return value, we do not need to join in D. Doing so would not hurt correctness, but there is no need because we are guaranteed to have joined prior to D.

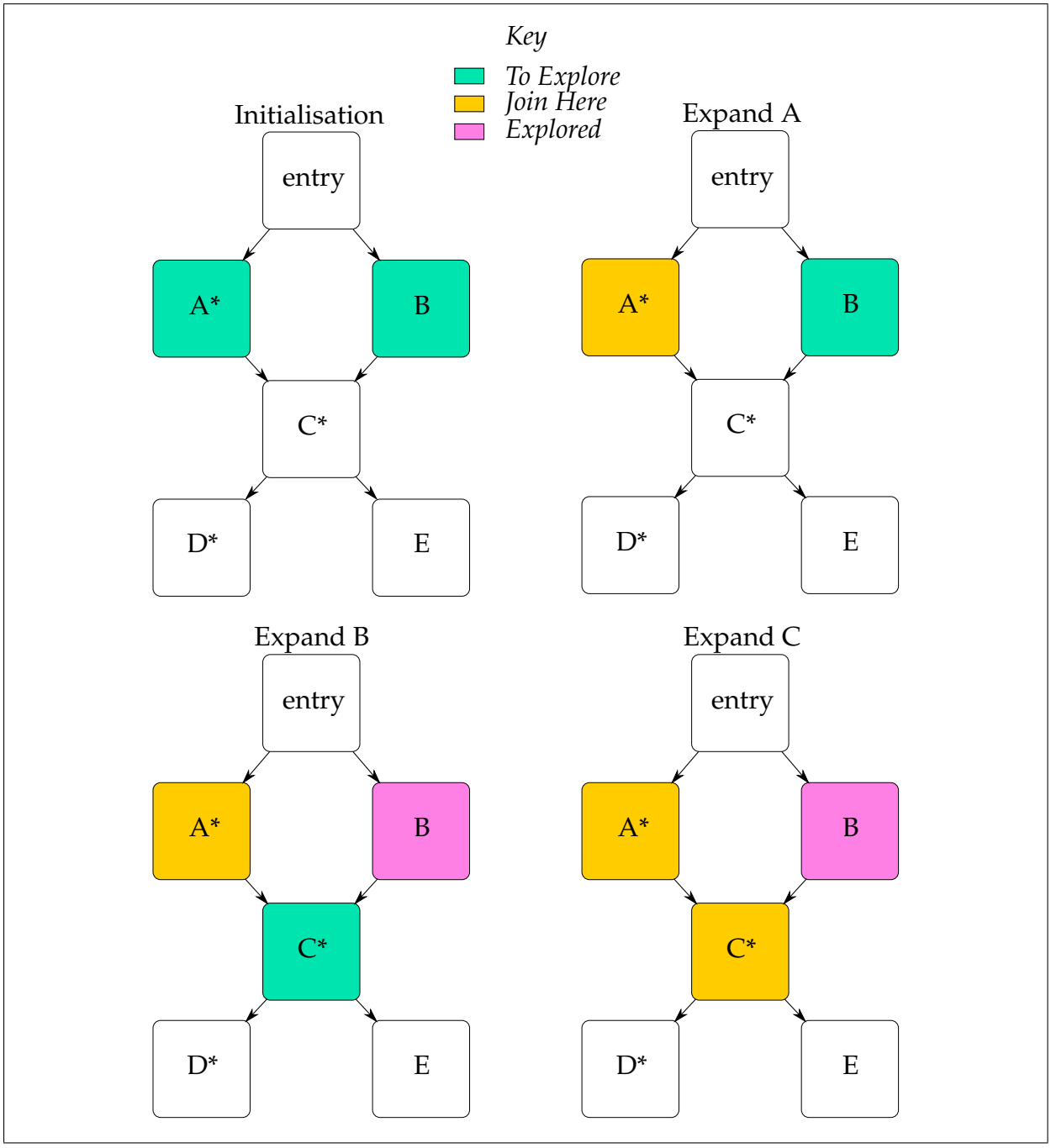

Figure 3.3: Algorithm 3.5 executing on a simple CFG.

3.4 The Decider Analysis Pass

At this point in the process, the Analyser has computed a lot of information about the program – which functions can be safely spawned, an approximation of their cost and where we can safely join with them, if they were to be offloaded. We *almost* have enough to make the final decision using the Profitability Condition (Figure 2.3) and decide which calls should be offloaded. The Decider will combine all the information we have to make this final decision before Hydra passes control to the Transformer.

3.4.1 The Missing Piece

To compute the Profitability Condition (Figure 2.3), we require $cost(caller)$, $cost(callee)$ and $cost(spawn)$, the second of which has already been computed by Profitability (§3.2), and the last is found ahead of time empirically (see §4.3). All that remains is to compute $cost(caller)$, the amount of work which can safely be done in parallel with the callee. As such, we require a heuristic which estimates the cost of the work between the spawn and the join(s).

The heuristic I use in the Decider is similar to h_2 (§3.2.3). h_2 was a refinement which included the cost of called functions. Since the Decider sees the output from Profitability, it already knows the costs for all functions which may be called.

An h_3-style heuristic (§3.2.4) was not used because there is added complexity to loop cost when we want to know the cost between *two points in the function* rather than the whole function.[6] Note that since h_3 is used by Profitability, it is also used in the Decider for the cost of function calls.

3.4.2 Deciding for 'Exactly-Once' Runtimes

On 'exactly-once' runtimes (§3.3.2), calculating $cost(caller)$ is simple, since the join is always in the same block as the call. All we have to do is apply $cost_3$ (§3.2.4) to each instruction between the spawn and the join. Let $range(S, J)$ be the set of all instructions between the spawn, S, and the join, J. Then, we can calculate the cost of the callee using the equation below, implemented by Listing 3.9.

$$cost(caller) = \sum_{I \in range(S,J)} cost_3(I).$$

3.4.3 Deciding for 'At-Least-Once' Runtimes

'At-least-once' runtimes (§3.3.3) are more complex, because the JoinPoints analysis may return multiple joins, all of which in different basic blocks to the spawn. We could

[6] In a function where the spawn and the join(s) are inside the body of some loop, h_3 will be an overestimate, because it does not see that the join will be reached before the loop's back edge.

```
int countInstructions(bb_citer first, bb_citer last,
                      const Profitability &Profit) {
  int count = std::accumulate(first, last, 0,
    [&](int runningTotal, const llvm::Instruction &i) {
      int extraInsts{ 1 };
      if (auto *ci = dyn_cast<CallInst>(&i)) {
        extraInsts += Profit.getFunStats(
                        *ci->getCalledFunction())->totalCost;
      }
      return runningTotal + extraInsts;
    });
  return count;
}
```

Listing 3.9: Calculating $cost(caller)$ for 'exactly-once' runtimes.

proceed by applying $cost_3$ to every possible instruction between the spawn and the join. However, this is likely to be an *overestimate* of the cost, and we generally want to avoid overestimating $cost(caller)$ since it is often the deciding factor in the Profitability Condition (Figure 2.3).

A conservative alternative used in Hydra finds the *shortest paths* to the joins. We can do this by building a *weighted graph*, where the nodes represent instructions of interest, and the weights represent the h_2 cost of the path. Then Dijkstra's Algorithm [11] can be used to efficiently compute the shortest paths. The only requirement of Dijkstra is that the graph must not contain negative weights, which is fine since h_2 is always non-negative.

Building the Graph

There is no graph infrastructure in the C++ Standard library, so instead I turned to Boost, a portable C++ template library. Boost has an implementation of Dijkstra's algorithm, so all the Decider needs to do is to design a custom graph based on the CFG which will yield the shortest paths to the join, as required.

The idea I used is to have a graph where the nodes correspond to certain instructions in the CFG: the call, join, entry and terminator instructions. Then, I link them together with positive weights for intra-block links, and zero weights for inter-block links. Finally, after Dijkstra has run, the required costs can be extracted by looking at the shortest paths to the nodes which correspond to join points.

Figure 3.4 demonstrates the four different kinds of node in the constructed graph from the CFG, with zero weight edges in green, and positive weight edges in blue.

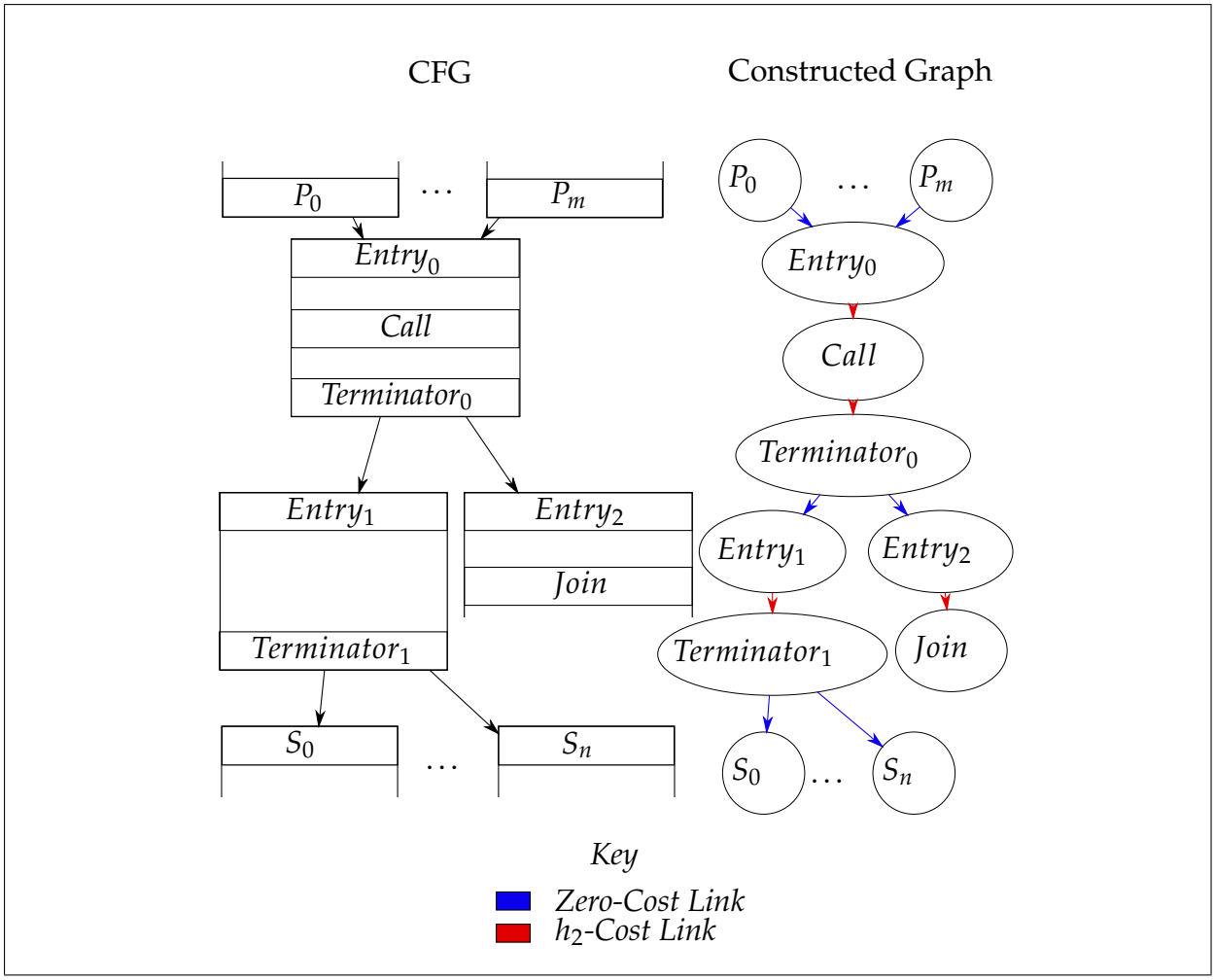

Figure 3.4: Illustration of the CFG to custom graph conversion.

Accumulating the Results

Now that we have the costs of the shortest paths to each join, the question remains as to how we combine all of these to estimate $cost(caller)$. Hydra supports three different methods. Denote the set of the shortest paths as *paths*, and the cost of a path, p, as $cost(p)$. Then, we can write down our various approaches for estimating $cost(caller)$.

There is the optimist's approach:

$$cost(caller) = \max_{p \in paths} \{cost(p)\}$$

The pessimist's approach:

$$cost(caller) = \min_{p \in paths} \{cost(p)\}$$

And the realist's approach:

$$cost(caller) = \frac{1}{|paths|} \sum_{p \in paths} cost(p)$$

Actually, the 'right' way to do this is by taking into account path probabilities, and then calculating $cost(caller)$ as an expected value, as below. However, calculating $\mathbb{P}(p)$ requires *Profile-Guided Optimisation*, which is one of the features Hydra might support in future work (see §5.3).

$$cost(caller) = \sum_{p \in paths} \mathbb{P}(p) cost(p).$$

3.5 The MakeSpawnable Transformation Pass

At this point in the process, the Analyser has completed. It has done the hard work, and Hydra knows which functions it wants to offload. However, there are a few interface concerns that it has to address before we can go ahead and replace the call with a request to spawn. A new function signature needs to be synthesised to conform with the spawning APIs of our supported runtimes, and it is the MakeSpawnable transformation's job to do this.

3.5.1 Statement of the Problem

Let us first consider the problems which MakeSpawnable must solve. Most of the these come from the Kernel Threads runtime (§2.2.1), but their solution allows simplification of the Thread Pool runtime (see §3.7).

All the functions that Hydra would like to spawn use *return values* to get their results back to the caller. The alternative would be to use an 'out-parameter', but this

is banned since Hydra forbids functions from using pointer arguments. This causes a slight problem: when we create an `std::thread` object with a function to execute, the return value is ignored. To get around this, we require the synthesised function to return via an out-parameter instead. I describe this mechanism in §3.5.2.

Another concern is that the function to be offloaded may have parameters of *user-defined type*. This a problem because `std::thread` deals with user-defined types using variadic templates, but the C++ template machinery is not available at the LLVM IR level of the compiler. The solution is to pass all arguments by `void*` pointer in the synthesised function, which is explained in §3.5.3.

3.5.2 Dealing with Return Values

As hinted above, we need to replace the return value of the original function with an out-parameter. Conceptually, this is a very simple transformation. Listing 3.10 shows how this transformation works. A convention used is to prepend '`_spawnable_`' to the name of a function when synthesising its spawnable version.

```
declare i32 @f()

define void @_spawnable_f(i32* %out) {
  %1 = call i32 @f()
  store i32 %1 i32* %out
  ret void
}
```

Listing 3.10: Synthesis of `_spawnable_f` from `f` in LLVM IR.

`f` returns an `i32`, which roughly corresponds to the C type `int` on most architectures.[7] What we need to do is call `f` within `_spawnable_f` and store its return value into the new out-parameter, `%out`.

That is all there is to it. It is up to ParalleliseCalls to ensure that this new out-parameter is used by previous users of the return value (see §3.6). Note that the synthesis used in Hydra will actually return by `i8*`, rather than the `i32*` used here, to deal with user-defined return types. We will see how this works in the next section.

3.5.3 Dealing with User-Defined Types

As previously mentioned, `std::thread` uses a variadic template to deal with functions of arbitrary type, but this machinery is unavailable to us. To solve this, we have a couple of options. To avoid unnecessarily complicating the Evaluation (§4), the second of these was chosen.

[7] Unlike C, LLVM IR does not denote differentiate between signed and unsigned types – it is down to the arithmetic instructions state whether they perform signed or unsigned operations.

- Hydra could emit a *list of template instantiations* which it would like someone to instantiate and link in. While this approach has its advantages, it is inconvenient and makes compilation more complex, especially since LLVM has no mechanism to do this.
- One could instantiate all the templates needed with `void*`,[8] and then have Hydra emit casts in the synthesised function. This does mean that primitive types suffer a penalty since they need to be placed on the stack rather than passed in registers.[9]

Casting is possible in LLVM IR with the non-emitting `bitcast` instruction, which will produce a new SSA value with the same bit pattern as the old value, but with a different type. The synthesised function needs to have one of these instructions for each parameter of the original function, plus one more for the return value. It also needs to load register arguments, so they can be passed by register to the original function. Listing 3.11 demonstrates this.

```
declare i32 @g(i16 %x,i64 %y)

define void @_spawnable_g(i8* %x, i8* %y, i8* %out) {
  %1 = bitcast i8* %x to i16*
  %2 = load i16* %1
  %3 = bitcast i8* %y to i64*
  %4 = load i64* %3
  %5 = call @g(i16 %2, i64 %4)
  %6 = bitcast i8* %out to i32*
  store i32 %5 i32* %6
  ret void
}
```

Listing 3.11: Synthesis of `_spawnable_g` from `g` in LLVM IR.

This is all MakeSpawnable has to do. Again, it is up to ParalleliseCalls to ensure that the arguments passed in at the call site are of type `i8*`, although the underlying type of the pointee must be correct to avoid undefined behaviour.

3.6 The ParalleliseCalls Transformation Pass

Just one hurdle remains before the baton can be passed on to later stages of the compiler, and that is the transformation itself. We will find that this is mostly straightforward, but there are a few difficulties which must be dealt with, and these are discussed ahead.

[8] Abstractly, `void*` is a C type which means 'a pointer to memory of unknown type'. LLVM IR has no `void*` type, so `i8*` is used instead.

[9] An industrial-strength implementation could make a special case of primitive types to avoid this penalty.

3.6.1 Adding the Runtime API to the Module

A lot of the work presented ahead is quite dependent on the target runtime in question. Recall that the two supported runtimes of this project are Kernel Threads and the Thread Pool. ParalleliseCalls needs to add the signatures for the APIs we require into the LLVM IR module so that we can emit the spawns and joins. Their implementation will be linked in later.

Kernel Threads API

For Kernel Threads, we need to construct a thread in place, then join with it and destroy it later. As such, the APIs that we care about are the Constructor, Destructor and Join functions. The C++ signature for these is given in Listing 3.12. The Constructor is defined using a variadic template, the difficulties of which were addressed in §3.5.3.

```
// Constructor
template<class Function, class... Args>
std::thread::thread(Function&& f, Args&&... args);

// Destructor
std::thread::~thread();

// Join
std::thread::join();
```

Listing 3.12: The `std::thread` API, used in the Kernel Threads runtime.

Thread Pool API

The API for the Thread Pool is given in Listing 3.13. The implementation details are described in §3.7. There are two API differences to Kernel Threads worth mentioning: first, the Thread Pool does not have a Destructor, and second, several overloaded Spawns are used rather than a variadic template, which alleviates the problems associated with them. We also require an `unsigned`, which is used at join-time to identify which tasks the Thread Pool needs to join with. We will see how this is addressed in ParalleliseCalls in §3.6.2.

Name Mangling

LLVM IR does not support function overloading. C++ front-ends get around this via a process called *Name Mangling*, where C++ function names are combined with the function's type to form a unique name. To ensure that this process is consistent, it is specified in an Application Binary Interface (ABI).

```
// Spawns
void spawn(unsigned task, void (*f)(void));
void spawn(unsigned task, void (*f)(void *), void *arg1);
void spawn(unsigned task, void (*f)(void *, void *),
           void *arg1, void *arg2);
//...

// Join
void join(unsigned task);
```

Listing 3.13: The Thread Pool runtime API.

Hydra adds only the necessary spawns to the module, but also does not impose any arbitrary restrictions on the maximum number of arguments an offloaded function may have. Algorithm 3.6 uses the Decider's output to calculate the arities of every offloaded function. It then uses GENERATECTORNAMES to generate the mangled signatures for each arity, which can be tailored to a specific ABI.

Algorithm 3.6 Generate the required constructors.

```
1: function GENERATECTORS(Decider)
2:                                          ▷ Calculate which Arities are Needed
3:     arities ← ∅
4:     for all F ∈ Decider.funsToBeSpawned() do
5:         arities ← arities ∪ {F.numArgs()}
6:     end for
7:                                          ▷ Generate all Required Signatures
8:     return GENERATECTORNAMES(arities)
9: end function
```

Hydra provides an implementation of GENERATECTORNAMES for the Itanium C++ ABI.[10] To conform with Itanium, Hydra assembles the constructor/spawn names from six parts, `s0`–`s5`, given in Figure 3.5. The non-zero argument constructor names are produced as all the `s` values concatenated together, except that `s2` and `s4` must be extended for each additional argument by `deltaS2` and `deltaS4` respectively. The zero argument constructor omits `s1` and `s4`. This is a little over-expressive for the Thread Pool, but using it meant I could use the same algorithm for both.

Algorithm 3.7 builds up the mangled names iteratively by extending `s2` and `s4` on each loop iteration. The name is then added to the returned set if the arity of that name is in the set of requested arities. Notice that the zero-argument constructor name is computed prior to entering the loop, as it is a special case.

[10]While the Itanium C++ ABI was originally written for Intel's Itanium CPU architecture, it has since been repurposed for many other architectures, like x86.

Kernel Threads:

```
     s0 = "_ZNSt6threadC2IRFv"
     s1 = "P"
     s2 = "v"
     s3 = "EJ"
     s4 = "RS1_"
     s5 = "EEEOT_DpOTO_"
deltaS2 = "S1_"
deltaS4 = "S4_"
```

Thread Pool:

```
     s0 = "_Z5spawnjPFv"
     s1 = "P"
     s2 = "v"
     s3 = "E"
     s4 = "S_"
     s5 = ""
deltaS2 = "S_"
deltaS4 = "S_"
```

Figure 3.5: s values for the two supported runtimes.

Algorithm 3.7 Generate all required Itanium-mangled ctor names.

function ITANIUMCTORNAMES(*arities*)
    $ctors \leftarrow \emptyset$
    ▷ Generate zeroth name.
    $name \leftarrow s_0 + s_2 + s_3 + s_5$
    for all $i \in \{0, 1, \ldots, max(arities)\}$ **do**
        ▷ Check if the i^{th} name is required.
        if $i \in arities$ **then**
            $ctors \leftarrow ctors \cup \{name\}$
        end if
        ▷ Generate the $i + 1^{th}$ name.
        $s_2 \leftarrow s_2 + deltaS_2$
        $s_4 \leftarrow s_4 + deltaS_4$
        $name \leftarrow s_0 + s_1 + s_2 + s_3 + s_4 + s_5$
    end for
    return *ctors*
end function

3.6.2 Dealing with the 'Task' ID in the Thread Pool

The Thread Pool API (Listing 3.13), requires an `unsigned` for spawning and joining so that the Pool can identify which tasks need to be joined with at join-time. It is perfectly safe to generate a random constant at compile-time and use that, since if two task IDs clash, we will simply join with tasks a little early in the Thread Pool and we only have a performance problem, *not* a correctness problem.

C++ provides facilities for generating fast, high quality random data. Hydra uses the included *Mersenne Twister* generator [12]. Mersenne Twister is fast, deterministic (which is useful for debugging) and extremely high-quality. Hydra uses `std::random_device` to seed the twister, which is designed to produce unpredictable (sometimes cryptographically secure) random data, but can be quite slow.

3.6.3 Switching the Uses of the Return Value

Once the API signatures have been added to the module, one difficulty remains – Hydra must replace all the previous uses of the return value with the new return value.

Loading the Return Value

Recall that the spawnable version of our function will get its return value to us by storing it into its out parameter when it has finished. As such, it is vital that we load it *after* joining with the function.

Hydra could load the value at the join(s), and swap the uses of the return value with this newly loaded value, but this becomes complex if there is more than one join. Fortunately, LLVM encourages passes to be liberal with the number of loads used, and allow later stages of the compiler (and even the hardware) to optimise redundant loads. This is the approach taken by Hydra.

Finding all the Uses of a Value

Hydra finds all the uses of the return value, and replaces them with a freshly emitted load of the offloaded function's out parameter. Rather than searching the CFG manually, LLVM provides a `use_iterator` to conveniently iterate over all the uses of an SSA value. The implementation is given in Listing 3.14. Within the transform on line 10, if the value is the call instruction, then it is using the return value and we swap it with the load generated on line 8. Otherwise, we return what we were given to the effect of not modifying the value.

3.7 The Thread Pool

Once the compiler has emitted the target code, sprinkled with spawn and join API calls, all that remains is to run it. For the Kernel Threads runtime, no more work is required,

```
1  void Hello::handleReturnValue(CallInst *ci,
2                                AllocaInst *retVal) {
3    std::vector<User *> oldUsersOfCall{};
4    std::for_each(ci->use_begin(), ci->use_end(),
5          [&](User *u) { oldUsersOfCall.push_back(u); });
6
7    for (User *u : oldUsersOfCall) {
8      auto *load = new LoadInst{ retVal, "retVal",
9                                 cast<Instruction>(u) };
10     std::transform(u->op_begin(), u->op_end(), u->op_begin(),
11         [=](Value *v) {
12           if (v == ci) {
13             return load;
14           } else {
15             return v;
16           }
17         });
18   }
19 }
```

Listing 3.14: Swapping uses of the return value for the offloaded function's out parameter.

since it builds off the existing `std::thread` API. However, for the Thread Pool, more implementation is necessary.

At the start of the program, the Thread Pool must initialise its worker threads, discussed in §3.7.1. In addition, the APIs come in two halves – we need to figure out what to do at spawn-time (§3.7.2) and join-time (§3.7.3).

3.7.1 Initialisation and Layout

The state used by the Thread Pool can be bounded by a constant, so it does not need to dynamically manage memory. Its layout is given in Listing 3.15. The macro `NUM_THREADS` is used to select the number of threads at compile-time. While using a `bool[]` is less space efficient than a `bitset` on most targets, it is required because each thread must be able to read and write to each `bool` independently, so they cannot live in the same byte. Note that C++11 guarantees that elements of an array can be synchronised independently, which means the compiler will add padding to these arrays to avoid false sharing if necessary.

The constructor used for initialisation is given in Listing 3.16. Workers are also signaled to stop through their `bool` in the `stops` array. The `atomic<bool>` in `hasJobs` is used for back-and-forth communication, asserting whether the worker has a job. As such, both of these should be initialised to *false*. `available` is used by the Thread Pool to record which threads are able to take on new work, and should be initialised to *true*.

```
constexpr unsigned numThreads = NUM_THREADS;

class ThreadPool {
  Job jobs[numThreads];
  std::mutex mutexes[numThreads];
  bool stops[numThreads];
  std::thread threads[numThreads];
  std::atomic<bool> hasJobs[numThreads];
  bool available[numThreads];
  std::mutex available_mutex;
};
```

Listing 3.15: Layout of the Thread Pool.

Finally, the `threads` need to be initialised with the address of their data and the '`do_work`' function, which is the worker's main loop.

```
ThreadPool::ThreadPool() {
  for (unsigned i = 0; i < numThreads; ++i) {
    stops[i] = false;
    hasJobs[i] = false;
    available[i] = true;
    threads[i] = std::thread(&do_work, &jobs[i], &mutexes[i],
                             &hasJobs[i], &stops[i]);
  }
}
```

Listing 3.16: The Thread Pool's constructor.

The destructor in Listing 3.17 does a proper clean-up. It may be faster to do a quick exit and leak all the threads, but this depends on the system. We need to lock and unlock each mutex so that the threads do not see a torn `stops` value. An `atomic<bool>` could have been used here instead.

3.7.2 Communicating Tasks to the Worker Threads

Listing 3.15 references a user-defined `Job` struct. This is used to communicate tasks to a thread. It contains the number of parameters for the task, a function pointer to the offloaded function, and pointers to the arguments. To offload a task to a thread, I initialise the thread's `Job` and set its `hasJob` to *true*, signalling that it has a job and its `Job` struct is now valid.

At spawn-time, we need to see if there are any available workers, and call the function locally otherwise. We do this by scanning the `available` array looking for a value of *true*.

```
ThreadPool::~ThreadPool() {
  // signal all threads to stop
  for (unsigned i = 0; i < numThreads; ++i) {
    mutexes[i].lock();
    stops[i] = true;
    mutexes[i].unlock();
  }

  // join on all threads
  for (auto &t : threads) {
    t.join();
  }
}
```

Listing 3.17: The Thread Pool's destructor

This is linear in the number of threads in the pool. An industrial-strength implementation may use a circular buffer to reduce the algorithmic complexity.

If there are no threads available, the function is called within the body of the spawn API. This means that Hydra's compile-time component does not need to emit code into the target to inspect whether a task was successfully spawned, as this is all handled by the API.

3.7.3 Joining with Worker Threads

At join-time, the pool matches the given taskID to all threads running a task with that ID. A key insight is that tasks must be spawned and joined by *the same thread*, so we can use thread-local storage to record which tasks are in-flight, and assigned to which worker-threads. The implementation is given in Listing 3.18. Again, it is linear in the number of active tasks, though this is at most the number of threads in the pool. The `remove_if` algorithm used on line 8 maintains `taskJobPairs`, which works by copying over elements which match the predicate.

3.8 Summary

In this chapter, I have covered all the components of Hydra which were implemented – the Analysis and Transformation passes and runtime-specific code generated. We have seen that the Analyser tries to find profitable parallelisation opportunities though static analysis, and then delegates the task of exposing the parallelism to a supported runtime to the Transformer. Finally, we have seen that the Thread Pool is an efficient implementation of a supported runtime.

In the next chapter, I will test Hydra using unit tests, microbenchmarks and real programs to asses the degree to which it succeeds on meeting its goals.

```
1  thread_local unsigned spawnCount = 0;
2  thread_local std::pair<unsigned, unsigned>
3  taskJobPairs[numThreads];
4
5  void join(const unsigned task) {
6    // join with and remove all jobs in taskJobPairs which
7    // match task
8    auto *end = std::remove_if(taskJobPairs,
9        taskJobPairs + spawnCount + 1,
10       [&, task](const pair<unsigned, unsigned> &p) {
11         const bool join{ p.first == task };
12         if (join) {
13           tp.join(p.second);
14         }
15         return join;
16       });
17   assert(end - taskJobPairs >= 0);
18   spawnCount = end - taskJobPairs;
19 }
```

Listing 3.18: The `join` API call in the Thread Pool.

Chapter 4

Evaluation

This chapter shows all that was done to establish that Hydra works as intended, but also how well the function offloading performs in practice. As Hydra is a compiler extension, unit tests are of high importance due to low tolerance to incorrectness and numerous corner cases. We will see how test cases were constructed (§4.2) and how the runtimes were evaluated for efficiency (§4.3). Finally, I apply Hydra to a real-world simulation problem, demonstrating that it successfully speeds up the completion by auto-parallelising it (§4.4).

4.1 Overall Results

In the Requirements Analysis (§2.3), requirements were specified for each of the three components of Hydra: the Analyser, Transformer and Thread Pool.

The Analyser is required to '*determine which functions are fit for spawning and [...] at which callsides of those functions offloading is profitable*'. The Analyser was identified as being of maximal importance for a successful project, so its goals were met at the beginning of the implementation, as described in §3.1–3.4. The result is a sequence of optimisation passes, which ultimately outputs which callsites should be offloaded, as well as where to join with the offloaded call.

The Transformer is required to '*communicate to a supported runtime that certain function calls [...] should be offloaded.*'. This was implemented after the Analyser, as described in §3.5 and §3.6. It is able to add the relevent API declarations into an IR module, and emit spawn and join API calls at the point of offload.

The Thread Pool is required to be a '*supported runtime of the Analyser and Transformer which performs function offloading using an Eager Thread Pool design*'. This was the last component of Hydra to be implemented, but it has nonetheless been effective, as described in §3.7.

4.2 Unit Tests

A variety of tests were designed to verify the correctness of each of Hydra's optimisation passes. These come in the form of an LLVM IR module which is designed to test each corner case of the algorithms discussed in §3. A sample of the tests is given in Appendix B.1.

These were all tied together with a batch script, which can run all the tests and compare the output with an expected output. A screenshot the script after running all tests successfully is given in Figure 4.1, and the script itself is given in Appendix B.1.1.

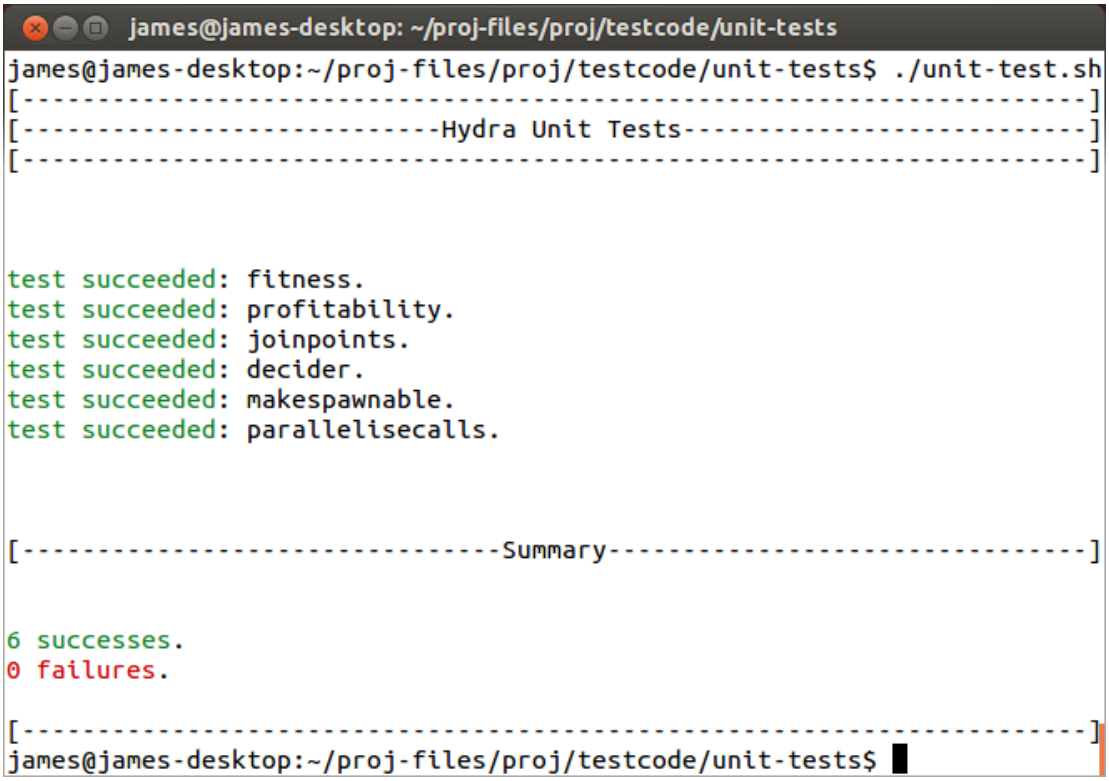

Figure 4.1: Screenshot of all unit-tests running with my shell script.

4.2.1 The Fitness Test

The Fitness pass (§3.1) has a variety of test cases, some of which should be declared fit, and others unfit. There are five aspects to test:

1. Functions which have *pointer arguments* must be declared *unfit*.
2. Functions which *explicitly reference global variables* must be declared *unfit*.
3. *Opaque functions*, for which we lack the definition, must be declared *unfit* – i.e. a library call without its source code.

4. Functions which *call unfit functions* must be declared *unfit*.

5. *All other functions* must be declared *fit*.

These are tested using a module with five functions, one for each aspect. The Fitness Test module and expected output is given in Appendix B.1.2.

4.2.2 The Profitability Test

For Profitability, I needed to test that the pass correctly calculates the h_3 heuristic presented in §3.2.4. To do this, I designed some IR modules, and computed h_3 for each by hand. Note that the value of h_3 depends on the output from LLVM's ScalarEvolution, so I needed to run the module through it to do the calculation.

In order to stress the corner cases of h_3, I made a function with a loop, a function with recursion, a function with a loop and recursion, and functions which call cheap and expensive functions.

4.2.3 The JoinPoints Test

To test JoinPoints, I designed two spawn points: one with the join inside the spawn block, and one where there were joins in multiple blocks. Note that when testing on the Kernel Threads runtime, both these examples should put the join in the spawn block, for reasons discussed in §3.3.2. For the Thread Pool runtime, we need to test that blocks with joins, blocks without joins, CFGs with loops, blocks with returns, and looping back around to the spawn block all work as expected.

4.2.4 The Decider Test

The correctness of the Decider is quite sensitive to the correctness of Profitability and JoinPoints, so this test is written under the assumption that those work correctly.

There are two key pieces of functionality to test in the decider:

1. $cost(caller)$ is being calculated correctly, using each of the three methods of combining all the results (§3.4.3).

2. The Profitability Condition (Figure 2.3) is being applied correctly.

The first of these cannot be tested directly, since the Decider does not remember $cost(caller)$ after its decision has been made. Hence, I adopted an indirect method to test this: I designed examples where the choice of combination method should change the decision made.

4.2.5 The MakeSpawnable Test

All we need to see here is that the MakeSpawnable pass is correctly synthesising spawnable versions of functions, armed with the correct types and bitcasts. We also need to check that it does not synthesise spawnable versions of functions that are not going to be parallelised.

4.2.6 The ParalleliseCalls Test

ParalleliseCalls has a fair bit of functionality that needs to be tested, which is enumerated below. The Module that was written tests all these features.

1. *Name mangling* is working correctly for the Itanium C++ ABI (§3.6.1).
2. Task IDs are being correctly generated for Thread Pool calls. Note that we have to use a constant seed for the generator to ensure that the test results are deterministic.
3. Appropriate `bitcasts` and `allocas` are being added at the spawn site.
4. The *return value* of parallelised calls is being handled properly.

4.3 Runtime Microbenchmarks

The Profitability Condition (Figure 2.3) has three quantities. Two of them are established through static analysis, and the third, $cost(spawn)$, is found empirically. Here I will show how this value was estimated for Hydra's two supported runtimes.

It is important to understand the units of $cost(spawn)$. Observing the Profitability Equation tells us that it must have the same units as the other two quantities, which were estimated using heuristics based on the number of *IR Instructions* in §3.2 and §3.4. Thus, the required units must also be number of IR instructions.

Since measuring IR instruction counts directly is difficult, I chose an indirect approach: I measured the time it took to execute a spawn followed immediately by a join, and then normalised the time to that taken to execute a simple increment operation. An increment requires three IR instructions, and we can thus derive an approximation of the number of IR instructions needed for a spawn/join from this measurement An example of the code which generated one of the test results is given in Appendix B.2.

The results are given in Figure 4.2. We see that the Thread Pool outperforms Kernel Threads, managing to join in $10\times$ fewer instructions.

The Thread Pool runtime tests were performed twice: once when the pool was empty (cost of a successful spawn and join), and once when the pool was at capacity (cost of a failed spawn and benign join). The results are coloured green and red respectively. As predicted in §2.2.2, the cost of a failed spawn is low, usually no more than 10 IR instructions, but this is perhaps not small enough to be ignored entirely, suggesting a possible refinement to the Analyser.

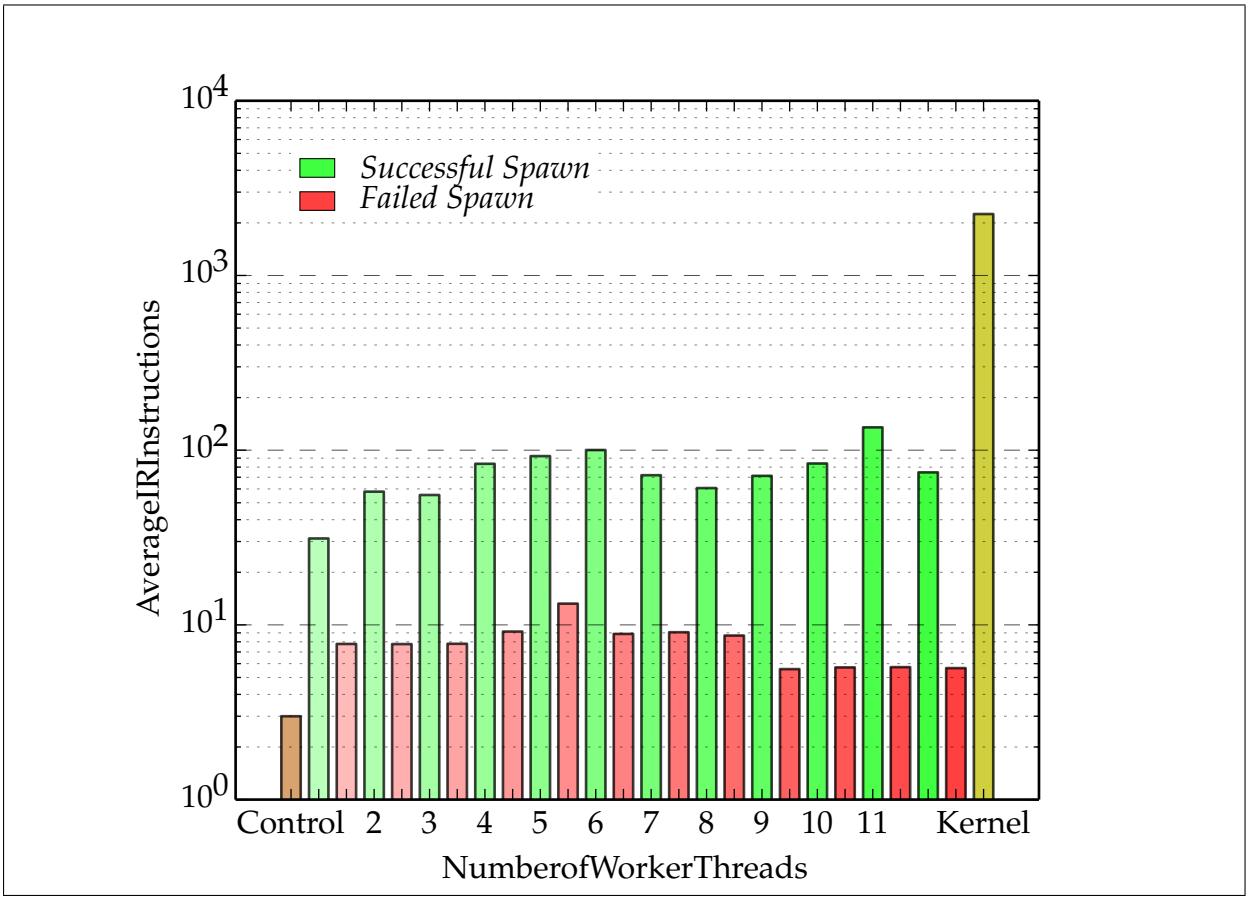

Figure 4.2: Mean number of instructions to spawn and join with tasks on Hydra's supported runtimes, compared to an increment operation. Green bars denote an empty thread pool, while red bars denote a pool at capacity. The y-axis uses a $\log_{10}$-scale.

In addition, the Thread Pool runtime tests were repeated for pools with different numbers of worker threads, but the differences are relatively minor. As discussed in §3.7, spawn and join are linear-time operations, but the constant factor seems too small to make a difference at 12 worker threads.

The tests produced quite a few anomalous results, probably due to a context switch occurring in between the timer points. As far as the Analyser is concerned, these context switches would have occurred anyway, so are not interesting when making an offload decision. In addition, the first spawn is consistently slower than subsequent spawns. This is best explained by the first test having a cache miss, hinting at another possible refinement to the analysis.

4.4 Performance Testing

To test Hydra's performance on a macroscopic level, I implemented a well-known algorithm which has a lot of parallelism available for extraction – n-body simulation.[1] However, my implementation is serial, i.e. it does not have *any* built-in parallelism whatsoever. It is up to Hydra to find and extract the parallelism.

N-body simulation is a numerical approach to the n-body problem: solving the trajectories of n gravitational bodies.[2] It is proven that, for $n > 2$, the n-body problem has no analytical solution, which makes simulation very popular.

My implementation of the algorithm, given in §B.3, iterates over each body, updating its velocity and position. Computing this update is quite costly, since we must loop over each other body in the scene and compute the inverse squared distance. Hence, Hydra is able to offload the update function.[3]

The code was compiled using Clang 3.4 targeting the x86-64 architecture. I decided to use optimisation level '`O1`' in order for the results to be reflective of my project's contribution to performance rather than Clang's optimiser. However, I felt that `O0` performance was unrepresentative.[4]

To expose parallelism from within the main loop, I used a technique called *loop unrolling*, where a loop is transformed such that the body of the new loop executes multiple iterations of the old loop. LLVM implements automatic loop unrolling and, with additional engineering effort, the project could hook into this for automatic loop parallelism extraction.

[1] Due to the fragile nature of n-body simulation, it also served as a good correctness test for the runtimes, since any errors would effect the observable output.

[2] `http://bit.ly/1iGtPzS`

[3] I do need to make a special case of these functions, since Fitness is rejecting them due to pointer arguments (§3.1). This is safe to do, since they do not modify the pointee (§2.1.2). With additional engineering work, I am confident that Fitness will be able to identify such use automatically.

[4] The Thread Pool itself was compiled at `O3`, since the Analyser's cost model relies on a highly optimised Thread Pool.

4.4.1 High-Confidence Tests

I order to be able to assert that the observed speedups are not by chance, I need a narrow confidence interval. Thus, the n-body experiment was run 1,000 times on my quad-core desktop machine.[5] I ran with a factor four unroll serially and with three worker threads (totalling at four threads, including the main thread).

Table 4.1 shows the arithmetic mean of the results, as well as a 99% confidence interval.[6] We see that there is no doubt that the project has resulted in a significant average speedup, albeit with increasing variance.

	Arithmetic Mean (99% confidence interval)	Standard Deviation
Serial	5.676736s $\pm$ 0.00004244s	0.0005201s
Parallel	2.908464s $\pm$ 0.001249s	0.01530s

Table 4.1: Result of 1,000 n-body simulations (100 bodies, 200 steps).

These results demonstrate a 1.952$\times$ speedup, compared to a theoretical maximum of 4$\times$. Later results actually show greater speedup when even more threads are used, but with wider confidence intervals (due to a lower number of repeated experiments).

4.4.2 Scalability Tests

The previous tests showcased significant speedup, but I wanted to see how these results scale to greater number of cores. I did two scalability tests – one on the same quad-core desktop machine as above, and another on a 48-core machine. I ran the simulation with Thread Pools of varying numbers of threads, as well as serially and with the Kernel Threads runtime (§2.2.1). The only difference between the programs run on each machine was the unroll factor (factor eight for quad-core, factor 48 for 48-core). Each test was run 50 times.

Quad-Core

The results for the quad-core machine are given in Figure 4.3. The times have been normalised by the mean serial time (5.1436s). The result differ from the high-confidence result above because of the more aggressive unroll factor.

We can see a linear speedup pattern from Serial to three worker threads. This is intuitive, since the pool is exploiting more resources with each additional worker thread. We also see that the standard deviations for these four experiments are very small, which also intuitive because there is little resource contention.

[5] This machine contains an AMD Athlon II X4 645 using the K10 microarchitecture clocked at 3.1 GHz and 4 GB of RAM.

[6] The serial confidence interval is much narrower because it has a smaller variance.

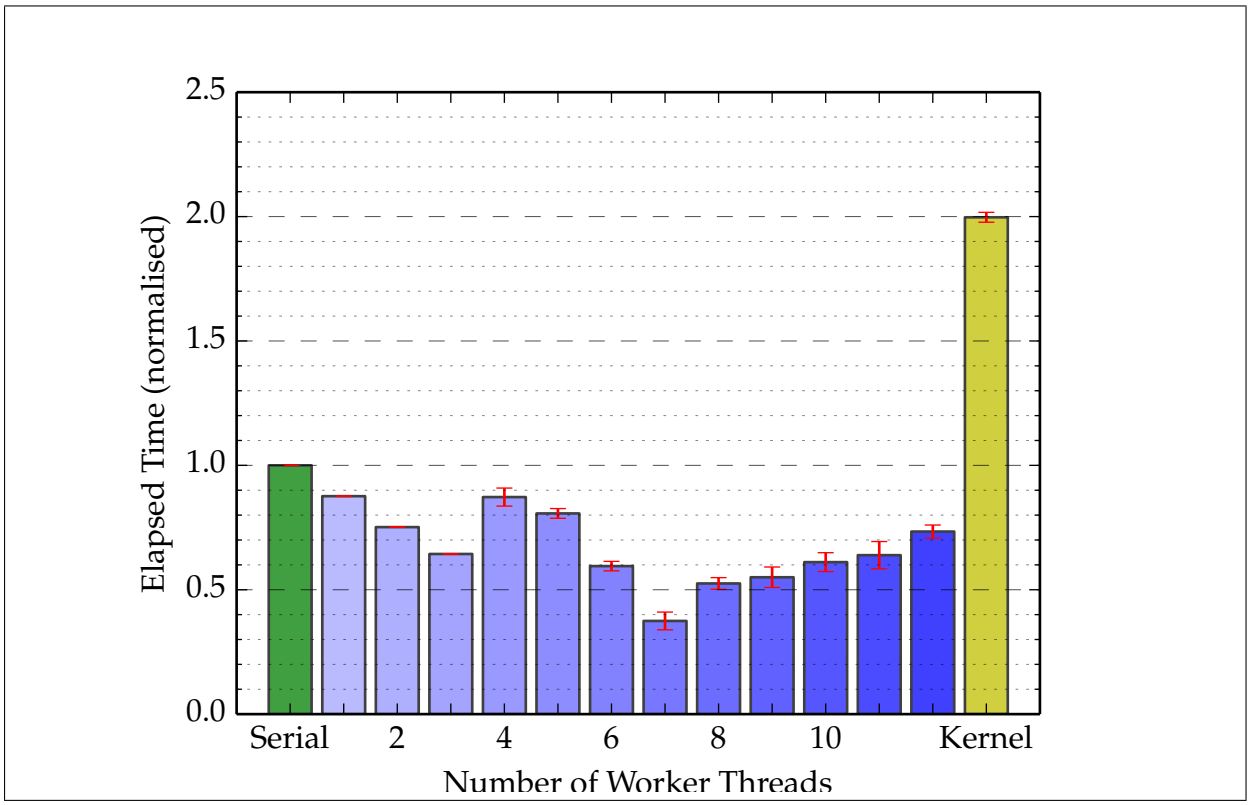

Figure 4.3: Means and Standard Deviations of 50 n-body simulations (100 bodies, 200 steps) on the quad-core machine.

From three to four worker threads, we see a sharp increase in both mean and variance of run-time. This is explained by the sudden resource contention that emerges – at three worker threads, no thread has to compete for CPU cores, while at four, we now have five threads competing for four cores.

Despite this, we see consistent improvements to mean performance from four to seven worker threads. This is because the worker threads execute in waves – the workers race for the four cores, and the remainder will have to wait. Since each worker has about the same amount of work, they all finish at roughly the same time, leading to a second wave of work. At seven worker threads, we extract the maximum from each wave, while fewer than seven results in an under-utilised second wave.

Beyond seven worker threads, performance gets worse. This is best explained by the loop unroll factor not being aggressive enough – at factor eight, it is unreasonable to expect more than eight parallel tasks at once. As such, we are creating more threads than are needed, which creates unnecessary pressure on the scheduler.

Finally, the Kernel Threads runtime performs truly abysmally. We see a 2× slowdown on Kernel Threads, and a large increase in variance. This is best explained by the huge overhead incurred by creating a new thread when needed, explored in the previous section, and this is done eight times on every iteration of the main loop. It should be noted

that I had to reduce what the Analyser believed to be the overheads of Kernel Threads in order to make it parallelise the code at all, since the actual overheads cannot balance the Profitability Condition (Figure 2.3).

48-Core

The results of the 48-core machine[7] are given in Figure 4.4. These have been normalised, just as above, by the mean serial time (7.74786s). Note that serial performance on is worse than that of the quad-core machine above due to the difference in base clock frequency (2.6 GHz vs 3.1 GHz).

We observe *linear speedup*, peaking with 6.33× speedup at about 47 worker threads. Past 47 workers, performance degrades as we add pressure to the scheduler for no real improvement in extracted parallelism. This linear speedup is a pleasing result, and negates the concerns of synchronisation costs diminishing the gains.

We do observe an almost periodic rise and fall in variance as the number of worker threads increases. This is best explained by the machine's Non-Uniform Memory Access (NUMA) memory design [13]. The machine's 128 GB of RAM is divided into four memory controllers attached to four sockets, where each socket contains 12 of the 48 cores. When a core attempts to access memory which does not reside in its socket, it must make a remote access request to pull in the data from the other socket. The result of this is that cores on the same socket share memory more efficiently than cores on different sockets.

Once the data has been fetched from a remote socket, it will be cached in the current socket's cache hierarchy. This means that, in our experiment, when lots of threads are allocated to the same socket, fewer remote accesses are required.[8] As the Linux scheduler is under a lot of pressure to schedule threads quickly, it will usually not consider NUMA cache affinity when scheduling threads onto cores. As a result, we experienced increased variance of performance due to the degree of chance in receiving a good scheduling. This also explains the pseudo-periodicity of variance.[9]

To my delight, even the Kernel Threads runtime performed quite respectably. I believe this because the Linux kernel itself is more able to create threads in parallel on the 48-core machine compared to my own desktop.

[7]The machine contains an AMD Opteron 6344 using the Abu Dhabi microarchitecture clocked at 2.6 GHz base, with 2.9–3.2 GHz turbo, and 128 GB of RAM. For more details, see `http://bit.ly/1oFV9z6`.

[8]When one core pulls data into the socket's cache, the other cores experience cache hits when they request the same data for reading.

[9]Remote accesses rise with number of threads, but plateau once all four sockets are used. As number of threads increases further, we experience more cache hits due to neighbouring cores pulling shared data into the socket's cache.

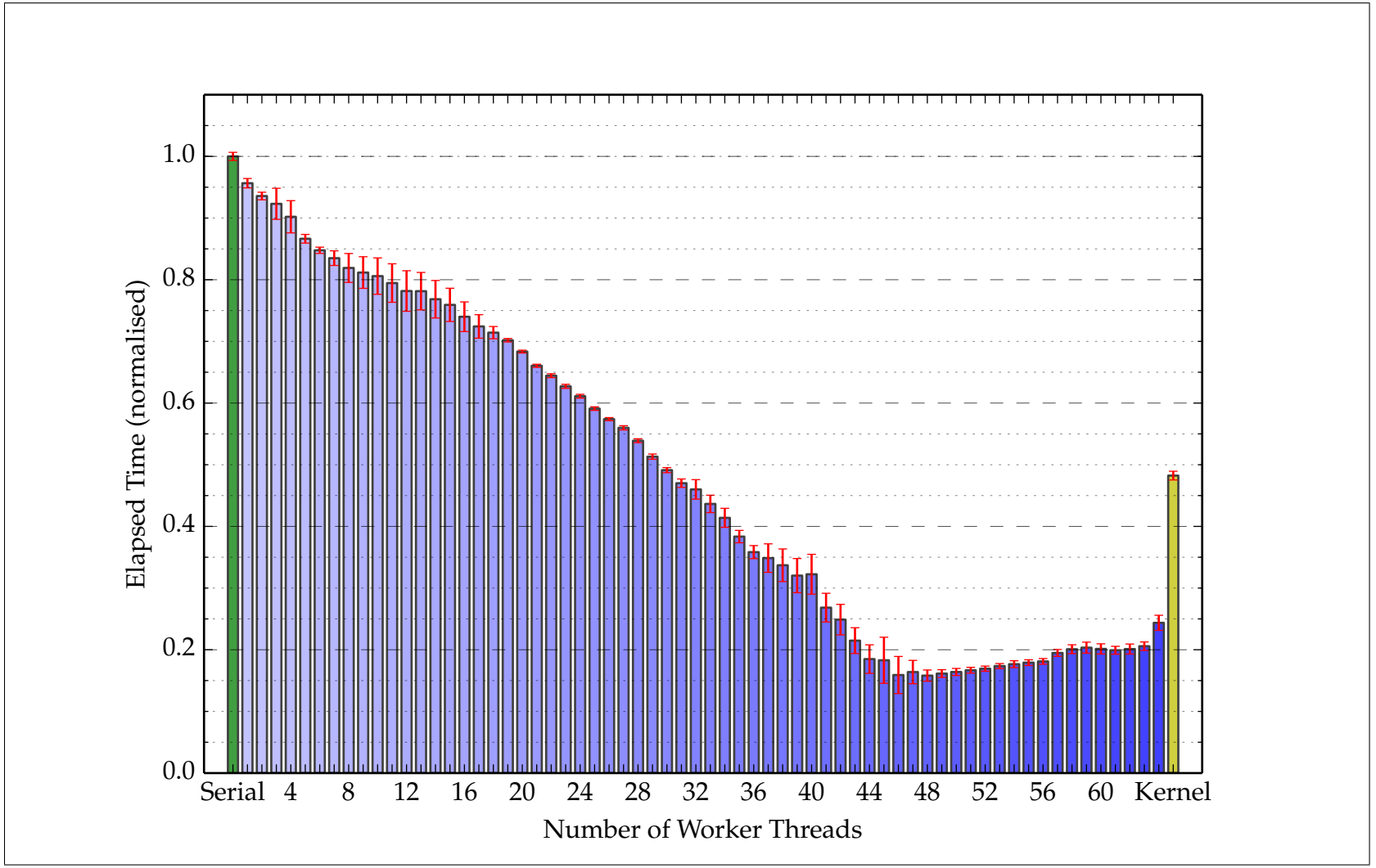

Figure 4.4: Means and Standard Deviations of 50 n-body simulations (100 bodies, 200 steps) on the 48-core machine.

Chapter 5

Conclusions

In this dissertation, I have explained all the preparation, implementation and evaluation I undertook in the development of Hydra.

When I decided to undertake this project, I desired to learn more about compilation, particularly optimising compilation, and parallelisation. Looking back, the project has certainly delivered: I had the opportunity to explore and experiment with leading compiler technology, and found myself inevitably learning a lot about related fields by osmosis. While Hydra is not a research project by nature, the difficult problems I faced have certainly increased my appetite for research in this field.

5.1 Results

As discussed in §4.1, Hydra has met all the requirements laid out in the Requirements Analysis (§2.3), which has been verified through careful testing (§4.2).

Hydra has even gained a few unplanned extensions along the way. In particular, many of the improvements to the Profitability (§3.2), JoinPoints (§3.3) and the Decider (§3.4) analysis passes were ultimately included, some of which were not conceived until a fair way into Hydra's implementation. In addition, the Thread Pool itself (§3.7), a major contributor to Hydra and its success, was a project extension. While Hydra's Thread Pool runtime may be a little primitive compared to some its industrial counterparts, I find there is elegance in its simplicity and portability which, when coupled with its impressive performance (§4.4), is very pleasing.

In addition to Hydra attaining its goals, I find that I have also progressed significantly as a direct result of this project. As well as further exposing me to an endlessly fascinating field, the project has given me many practical skills that I am sure will be transferred to my future career. I have gained greater appreciation of the discipline and effort required to build a sizeable piece of software, on top of the knowledge of many new tools, and even advancing my skills in technical writing.

5.2 Lessons Learnt

Perhaps it was naïve of me to suspect that I could successfully create a fully automatic function paralleliser when little of that nature currently exists. In spite of this, I have managed to create something which is completely successful in what it sets out to achieve, and even though I have learnt of the perils of underestimating the difficulty of a problem at the outset, maybe it was this naïvety that has made Hydra as successful as it is.

5.3 Future Work

While I managed to add more extensions into Hydra than I could have thought possible when proposing the project, the complexity of the problems faced mean that there is inevitably more that could have been done. As such, I am planning to ultimately include the features given below in a future iteration of Hydra. I am also planning to release the source code for others to download and peruse.

- As Hydra's implementation neared its completion, the value of *Profile Guided Optimisation* (PGO) became clear to me. The static analysis techniques used in Hydra's Analyser will always have limitations. In particular, it is usually impossible to reason about the behaviour of program inputs in an unsupervised manner. PGO empowers us with this knowledge, which is immensely valuable. In my mind, however, it is clear that PGO is not a replacement for static analysis – in fact, I find the two to be complementary.
- Support for *exceptions* – first within in the program, but not offloaded functions, and then also in offloaded functions.
- Allowing functions with *pointer arguments*, in some limited cases where the analysis is tractable, e.g. read-only functions.

Bibliography

[1] David W. Wall. *Limits of Instruction-Level Parallelism* Palo Alto, CA, USA, 1992.

[2] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. *Cilk: An Efficient Multithreaded Runtime System* MIT Laboratory for Computer Science, 1995.

[3] Dagum, L.; Menon, R. *OpenMP: an industry standard API for shared-memory programming.* Computational Science & Engineering, IEEE, 1998.

[4] Simon Campanoni, Timothy Jones, Glenn Holloway, Gu-Yeon Wei and David Brooks. *The HELIX Project: Overview and Directions* Harvard University, Cambridge, USA and University of Cambridge, Cambridge, UK, 2012.

[5] Dorit Naishlos. *Autovectorisation in GCC.* GCC Developers' Summit, 2004.

[6] Tobias Grosser, Hongbin Zheng, Raghes Aloor, Andreas Simbürger, Armin Größlinger and Louis-Noël Pouchet. *Polly – Polyhedral optimization in LLVM.* Universität Passau, Sun Yat-Sen University, Indian Institue of Technology and Ohio State University, 2011.

[7] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smtih, Anil Madhavapeddy and Steven Hand. CIEL*: a universal execution engine for distributed dataflow computing.* University of Cambridge Computer Laboratory, 2010.

[8] Chris Lattner and Vikram Adve. *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation.* Palo Alto, CA, USA, 2004.

[9] H. Hinnant, L. Crowl, B. Dawes, A. Williams, J. Garland, et al. *Multi-threading Library for Standard C++ (Revision 1).* ISO/IEC, 2008.

[10] ISO/IEC 9899. *TC2 Committee Draft.* ISO. 2005. pp. 108–112.

[11] Dijkstra, E. W. *A note on two problems in the connexion with graphs.* Numerische Mathematik, 1959.

[12] Makoto Matsumoto and Takuji Nishimura. *Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator.* Keio University/Max-Planck-Institut für Mathematik, 1998.

[13] Nakul Manchanda and Karan Anand. *Non-Uniform Memory Access (NUMA).* New York University, 2010.

[14] Jeffrey Dean and Sanjay Ghemawat. *MapReduce: simplified data processing on large clusters.* ACM, New York, NY, USA, 2008.

Appendix A

C++11, the 2011 standard for C++

When I first learnt C++, I was vaguely aware of a new 'C++0x/11' standard. However, at the time, compiler support was minimal, so I did not end up learning it. However, just before I started writing the proposal for Hydra, two open-source compilers, Clang and GCC, became C++11 compliant. This motivated me to start reading about the new standard and shortly after, I made the decision to use these extensions in this project. Below is a list of new language features used in Hydra. In addition, several of the new libraries were used, given in §2.6.2.

- Move Semantics.
- Generalised constant expressions with `constexpr`.
- Uniform initialisation syntax and initialiser lists.
- Type inference with `auto`.
- Range-based `for` loops.
- Lambda expressions.
- Null pointer constant with `nullptr`.
- Strongly typed enumerations with `enum class`.
- Thread-local storage with `thread_local`.

Appendix B

Evaluation Code

Here, I will provide some code used to perform the Evaluation. I will try not to include near-duplicate examples, restricting myself to a representative bit of code for each conceptual component of the evaluation.

B.1 Unit Test Code

B.1.1 Testing Batch Script – `unit-test.sh`

This script was used to run all the unit-tests to quickly identify any regressions in the code. It is given in Listing B.1.

```
#!/bin/bash

failures=0
successes=0
failues=0
red='\e[0;31m'
green='\e[0;32m'
default='\e[0m'

success() {
  echo -e "${green}test succeeded${default}:\t$1".
  successes=$(($successes + 1))
}

failure() {
  echo -e "${red}test failed${default}:\t${1}."
  failues=$(($failues + 1))
}

newline=[---------------------------------\
---------------------------------------]

echo $newline
```

```
echo [----------------------------Hydra Unit\
  Tests---------------------------]
echo $newline
echo
echo
echo

# run for each analysis pass
for a in fitness profitability joinpoints decider; do
  opt -load ~/proj-files/build/Release/lib/Tests.so \
    -test-module-${a} -o test-module-${a}.bc blank.bc
  opt -load ~/proj-files/build/Release/lib/Analyses.so -$a \
    -analyze test-module-${a}.bc > test-$a-output
  if cmp test-$a-output test-$a-expected
  then success $a
  else failure $a
  fi
  rm test-module-${a}.bc test-$a-output
done

# run for each transformation pass
for t in makespawnable parallelisecalls; do
  opt -load ~/proj-files/build/Release/lib/Tests.so \
    -test-module-${t} -o test-module-${t}.bc blank.bc
  opt -load ~/proj-files/build/Release/lib/Analyses.so \
    -load ~/proj-files/build/Release/lib/Transforms.so -$t \
    test-module-${t}.bc -o test-module-${t}-after.bc
  if cmp test-module-$t-after.bc test-module-$t-expected.bc
  then success $t
  else failure $t
  fi
  rm test-module-${t}.bc test-module-$t-after.bc
done

echo
echo
echo
echo [---------------------------------Summary\
--------------------------------]
echo
echo
echo -e "${green}${successes} successes${default}."
echo -e "${red}${failues} failures${default}."
echo
echo $newline
```

Listing B.1: Unit Test Batch Script

B.1.2 Test Sample

Here, I provide the test module and expected output for one of the unit tests, the Fitness Test, as Listings B.2 and B.3 respectively. For more information, see §4.2.1. Note that, since Fitness is internally using associative data-structures, the order of output is deterministic but not necessarily alphabetical or the same as input order.

```
@glob = global i32 42

define i32 @pointerArgs(i32*) {
  ret i32 0
}

define i32 @refsGlobal() {
  %loadedGlob = load i32* @glob
  ret i32 0
}

declare i32 @opaque()

define i32 @callsUnfit() {
  %1 = call i32 @refsGlobal()
  ret i32 0
}

define i32 @noneOfTheAbove() {
  ret i32 0
}
```

Listing B.2: The Fitness test module.

Listing B.3: Fitness test expected output.

```
Printing analysis 'Function Fitness for Spawning Analysis':
Printing info for 5 functions
The function refsGlobal() is Unknown
The function opaque() is Unknown
The function callsUnfit() is Unknown
The function pointerArgs() is Unknown
The function noneOfTheAbove() is Functional
```

B.2 Runtime Testing – `poolfailuretest.cpp`

Listing B.4 uses special processor instructions to measure operations with a very high precision. The test below also spawns some worker threads prior to the timings being taken so that the cost of a failed spawn can be measured.

```c
#include <stdio.h>
#include <stdint.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>

#include "../../threading/ThreadPool.h"

static __inline void
do_cpuid(unsigned int ax, unsigned int *p)
{
    __asm __volatile("cpuid"
            : "=a" (p[0]), "=b" (p[1]), "=c" (p[2]), "=d" (p[3])
            :  "0" (ax));
}

static __inline uint64_t
rdtsc(void)
{
    uint64_t rv;

    __asm __volatile("rdtsc" : "=A" (rv));
    return (rv);
}

uint64_t do_rdtsc(uint64_t ticks) {
    do {
        unsigned int __regs[4];
        do_cpuid(0, __regs);
        ticks = rdtsc();
    } while (0);
}


int main() {
  // exhaust all worker threads
  for (int i = 0; i < NUM_THREADS; ++i) {
    spawn(0u, []{});
  }

  // run the tests
  for (int i = 0; i < 100000; ++i) {
    uint64_t ctr1 = do_rdtsc(1);
    spawn(42u, []{});
    uint64_t ctr2 = do_rdtsc(1);
    join(42u);
    uint64_t ctr3 = do_rdtsc(1);

    printf("%ju\t%ju\n", (ctr2 - ctr1), (ctr3 - ctr2));
  }
```

```
  // join with the prior tasks
  join(0u);
}
```

Listing B.4: Measuring the cost of a failed spawn and benign join in the Thread Pool.

B.3 The *n*-body simulation – `nbody.cpp`

Listing B.5 gives my implementation of the *n*-body simulation. Notice that it is completely serial. Unrolling of the main update loop was used to allow Hydra to extract parallelism

```
#include <cassert>
#include <functional>
#include <random>

#include "nbody.h"

using namespace std;

vector<Body> GenerateBodies(const int numBodies, const int seed) {
  assert(numBodies > 0);

  vector<Body> ret;
  ret.reserve(numBodies);

  uniform_real_distribution<double> dist(0.0, 100.0);
  mt19937 twister(seed);
  auto genVal = bind(dist,twister);

  for (int i = 0; i < numBodies; ++i) {
    ret.emplace_back(i, genVal(), genVal(), genVal());
  }

  return ret;
}

vector<Body> Simulate(vector<Body> bodies, const int numSteps) {
  const int numBodies = bodies.size();
  vector<Body> nextBodies(bodies);

  for (int i = 0; i < numSteps; ++i) {
    Body *currBodiesPtr = bodies.data();
    Body *nextBodiesPtr = nextBodies.data();

    for (int j = 0; j < numBodies; ++j) {
      // compute forces and update next frame
      auto nextX0 = computeXAccel(j, currBodiesPtr, numBodies);
      auto nextY0 = computeYAccel(j, currBodiesPtr, numBodies);
      updateBody(nextX0, nextY0, currBodiesPtr[j], nextBodiesPtr[j]);
```

```
    }

    swap(bodies, nextBodies);
  }

  return bodies;
}

float computeXAccel(const int bodyID, const Body *bodies, const int size)
    {
  const float currMass = bodies[bodyID].mass;

  float xForce = 0.0f;

  for (int i = 0; i < size; ++i) {
    if (i == bodyID) continue;

    const float xDistance = myClamp(bodies[i].xPos - bodies[bodyID].xPos,
        0.1f);
    const float yDistance = myClamp(bodies[i].yPos - bodies[bodyID].yPos,
        0.1f);
    const float distanceSquared = xDistance*xDistance + yDistance*
       yDistance;
    const float distance = mySqrt(distanceSquared);
    const float force =
        G * bodies[bodyID].mass * bodies[i].mass / distanceSquared;
    xForce += force * xDistance/distance;
  }

  return xForce/currMass;
}

float computeYAccel(const int bodyID, const Body *bodies, const int size)
    {
  const float currMass = bodies[bodyID].mass;

  float yForce = 0.0f;

  for (int i = 0; i < size; ++i) {
    if (i == bodyID) continue;

    const float xDistance = myClamp(bodies[i].xPos - bodies[bodyID].xPos,
        1.f);
    const float yDistance = myClamp(bodies[i].yPos - bodies[bodyID].yPos,
        1.f);
    const float distanceSquared = xDistance*xDistance + yDistance*
       yDistance;
    const float distance = mySqrt(distanceSquared);
    const float force =
        G * bodies[bodyID].mass * bodies[i].mass / distanceSquared;
    yForce += force * yDistance/distance;
```

```
  }

  return yForce/currMass;
}

inline void updateBody(const float xAccel, const float yAccel, const Body
    &curr,
                       Body &next) {
  // update velocity
  next.xVel = curr.xVel + xAccel * step;
  next.yVel = curr.yVel + yAccel * step;

  // update position
  next.xPos = curr.xPos + next.xVel * step;
  next.yPos = curr.yPos + next.yVel * step;
}
```

Listing B.5: The *n*-body simulation used, prior to loop unrolling.

Appendix C

Project Proposal

Automatic Parallelism Across Multiple Threads and More Using LLVM
Part II Project Proposal

James Chicken
jmc223

25th October 2013

C.1 Introduction and Description of the Work

Concurrent and distributed systems have become a ubiquitous part of the modern computing landscape. Clock speeds have ceased following Moore's Law for some time now, plateauing at around 3 GHz. As a result, we no longer live in a world where we can write simple serial software and expect it to run twice as fast every couple of years. The path to faster software is clear: exploit more parallelism.

This is, of course, far from a new revelation: programmers have been writing parallel code for decades. However, in that time, we have discovered that writing parallel code is far from simple – many subtle bugs that are difficult to find creep into and thrive in our parallel code, only rearing their ugly heads at the most inconvenient of times. Even if we can manage to write correct parallel code, there is still no guarantee that it will actually

run faster than a simple serial solution. And what about all that legacy code, which we used to enjoy a 'free' speed increase in every few years? The software engineering effort to parallelise all that legacy code is enormous – and there's still no guaranteed speed up after all that!

Perhaps the responsibility of parallelisation falls on our compilers, then? In this project, I aim to explore that point of view. Can we generate fast concurrent object code from basic serial source code? I believe that speedups are indeed possible using automatic methods. I seek to discover what the degree of that speedup might be.

C.2 Starting Point

I will be basing my project on the LLVM compiler infrastructure [8]. LLVM can be thought of as a compiler back-end, featuring an intermediate representation (LLVM IR), architecture-independent optimiser (`opt`), code generator for various targets, and an interpreter/JIT compiler. I will be working with LLVM IR as the language for my compiler transformations. I will use the pre-existing LLVM 'pass' framework, which allows my project to be dynamically loaded by `opt` and be used in conjunction with other numerous optimisations.

Since the interface to the pass framework is written in C++, this will also be the programming language I use. I consider myself to be fairly confident in C++, but I am not familiar with LLVM, so I will need to familiarise myself with LLVM's classes and interfaces. I will use Clang as my C++ compiler (which is part of the LLVM project) and will write to the ISO C++11 standard.

I also intend to use C++11's `std::thread` [9] as the threading library for thread-level parallelism, since LLVM does not feature a library of its own and I would prefer a portable solution over a target-specific one, like `pthreads`.

C.3 Substance and Structure of the Project

C.3.1 Substance

The first transformation I aim to support is the deferral of certain function calls to a separate thread. The key piece of analysis required to make this transform correctness-preserving is knowing what state a function is allowed to change.

For functions written in a functional style, this is fairly straightforward. For instance, in the function `float sqrt(float x)`, if we can prove that `sqrt` does not modify global or static variables, all that we are really concerned with is the return value. All that is required here then is that the caller synchronises with the thread before using `sqrt`'s

return value. Consider:

```
void f(float x) {
  float y = sqrt(x);

  //...

  std::cout << y;
}
```

If we assume that `//...` does not use `y`, we can conceptually transform this to:

```
void f(float x) {
  float y;
  std::thread t(sqrt, x, y);

  //...

  t.join();
  std::cout << y;
}
```

This by-hand source transformation demonstrates the nature of the transformation rather than the implementation[1] – my project will be working with LLVM IR rather than C++.

There are some subtleties worth mentioning with this simple example. Firstly, `t` is not calling the same `float sqrt(float x)` from before, but actually a new overload `void sqrt(float x, float &y)`. This is because `std::thread` discards any return values upon completion, so we replace `sqrt`'s final return statement with an assignment to `y` instead.

Secondly, this new code is not exception safe. If `//...` throws before `t` is finished, `t`'s destructor will call `std::terminate`. This is bad news if `f` is not known to be `noexcept`[2], in which case we need to do the transformation with more care.

The primary goal of this project is doing such transformations whilst preserving correctness. However, there is little point in doing such transformations if the cost of synchronisation is large compared to the gained parallelism. For instance, consider the case above where `//...` is empty, or results in no emitted code. Then we have introduced a lot of hassle in setting up a thread and joining with it for no real benefit compared with the current thread simply executing the function. Doing this kind of cost-benefit analysis with sophistication might involve a compile-time heuristic and/or profiling.

[1] However, it is worth noting that `std::thread` is the threading library I will be using.

[2] Note that LLVM IR's `nounwind` corresponds to C++11's `noexcept`.

Also, better results will be achieved if this optimisation is performed after certain other LLVM optimisations, such as unreachable/dead code elimination.

This example is also a little comical too, since it's likely that the cost of `sqrt` is vastly smaller than the overhead of spawning a new thread. This is something that the cost-benefit analyser will have to take into account. It may be also worth considering alternatives to naïvely spawning a thread for each transformation – in a large program, having a pool of worker threads may prove more effective. This would give the cost-benefit analyser more slack, since the thread overhead per function call is lower, but also add a slow-down to program initialisation.

Functional-style calls are the easiest to get working correctly. However, I would also like to try and parallelise other kinds of functions. Some of these will only apply to the more imperative or object-orientated LLVM front-end languages, making them potentially less attractive. Here are the 'other kinds' of functions, listed roughly in order of difficulty:

- *Functional-style* (as described above).
- *'Almost functional'*. This is my term for functions which pass all their arguments by value, but may write to, or read from, global variable(s). E.g. in the C standard library, `sqrt` will write to `errno` if its argument is negative.
- *Const-reference/pointer*. These functions pass their their arguments by reference or pointer, but promise not to modify them, usually to avoid copying. This is very common in pre-C++11 C++.[3] Care must be made to ensure the caller doesn't modify the variable before the callee has had a chance to read it.
- *Imperative-style*. These functions pass their arguments by non-const references or pointers and modify the values.[4] Very similar to const-reference functions, except we now also have to make sure the callee doesn't ruin variables for the caller.
- *Const member functions*. This would allow calls such as `list::find()` to be parallelised. It should be possible for multiple const member functions of a class to execute in parallel, but a read-write mutex may be required to prevent a non-const member from executing.
- *Non-const member functions*. Very difficult to get speedups automatically, but will be interesting to investigate what can be done.

C.3.2 Structure

The project is divided into the following main sections:

[3] The C++11 standard made numerous changes to improve the efficiency of value semantics.

[4] C code written before the `const` keyword was introduced is actually not an issue, because LLVM's dataflow analysis will notice if variables are 'de facto `const`' and mark them as such.

1. Background study of the key problems and investigation of suitable algorithms – for instance, what makes a function suitable for deferral, and what care needs to be taken to preserve correctness.

2. Familiarisation of the LLVM project, particularly LLVM IR and the LLVM pass infrastructure, for building optimisation passes. I also need to familiarise myself with the various new tools I will be using (`make`, `git`, `latex`). I am already familiar with C++, but need to familiarise myself with C++11's `std::thread` as well. Planning implementation and test strategies.

3. Development and testing for the bulk of the code. This will involve mostly what is listed in the Substance section, and also any Extensions listed below, depending on my progress.

4. Evaluation using various sample programs, which I will write. These will feature some 'embarrassingly parallel' snippets, as well as some less easily parallelised ones. I can give quantitative evaluation in the form of 'before and after' profiling results, as well as in-depth explanations as to why my project was able or unable to transform some sample program.

5. Writing the Dissertation.

C.3.3 Extensions

In addition to the above, I have some objectives which are not critical to the success of the project, but would nonetheless make for an interesting addition to an already working implementation.

Control Flow Parallelisation

Parallelising function calls is nice, particularly since modern style guides recommend use of a large number of small functions over arbitrary control flow. However, much legacy code has been written which does not conform to this modern design philosophy, and the functions at the bottom of the call stack often still use loops to do their work, rather than recursion. For this reason, parallelising more general control structures would be an interesting extension.

One way I could do this is by executing separate cycles of a loop on separate threads. This can be used to speed up some common 'embarrassingly parallel' problems, such as matrix multiplication.

Another thing I could do is to try and parallelise more general basic blocks. Basic blocks make up the bulk of an LLVM IR program – they consist of blocks of code which contain no interesting control flow. It would certainly be interesting to analyse the flow of basic blocks though a program, attempt to find the critical paths though the block flow-graph, and discover which blocks can be executed in parallel.

Distributed Systems Parallelism

In the code written in my example above, one might notice how we generated code which executed the function on an `std::thread`. If this transformation was written in a sufficiently general manner, it should be possible to instead generate code which can be executed on a remote machine in a distributed system. The code could be generated to run on a MapReduce system [14], for instance. However, MapReduce is not the most natural choice, since it takes JVM code as input. While it is possible to convert LLVM IR into JVM, this does not fix the problem of library code, which is likely to be linked in object form. A better option might be to use CIEL [7], a Cambridge project which can work with a binary.

This does present some interesting problems, though. I would now have the option of running a parallelised piece of code on a different thread locally, or remotely on another machine. Choosing between them will likely involve extending the heuristics/profiling mentioned earlier. One way of doing it heuristically would be to analyse the amount of computation vs memory access in the code – if the code was heavy on computation, it may make more sense to run it locally rather than incur the remote overheads. On the other hand, lots of memory accesses may pollute the CPU's cache, so it may be worth offloading memory intensive work elsewhere. Such a heuristic should be backed up by profiling results. It may even be possible to perform this selection at run-time.

C.4 Success Criterion

To demonstrate, through well-chosen examples, that I have implemented one or more working LLVM passes which can analyse function calls in an LLVM IR program to determine whether it would be possible and profitable[5] to parallelise the call, and then generate the necessary transformations to execute that function on a different thread.

C.5 Timetable and Milestones

Michaelmas Term

Weeks 3 and 4

Study the tools to be used: `make` and `git`. Examine in detail the LLVM system, particularly the LLVM IR, and LLVM's pass infrastructure. Become familiar with `std::thread`, particularly its edge-case semantics.

Deliverables: A git repository for the project, structured to work with LLVM's Make system, and including some working example passes including cases which use `std::thread`.

[5] to some degree of accuracy

Weeks 5 and 6

Further study of the problem in question – gaining understanding of what must be done to preserve correctness when a function is parallelised, and how to compute the analysis results needed to prove safety and profitability. Consult with supervisor to ensure my understanding is sound. Begin writing code to do the required analysis.

Deliverables: The framework of an LLVM analysis pass, which can annotate function calls that it deems sensible to be parallelised.

Weeks 7 and 8

Test and polish the analysis pass above. Write and test a transformation pass to carry out the parallelisation transformations. If progress is going well, begin planning and study for the project extensions by gaining familiarity with CIEL and studying the loop parallelisation problem.

Deliverables: A tested and reasonably efficient analysis pass and transformation pass for deferring function calls.

Christmas Holiday

Begin planing for the evaluation – design small test programs which can be used for 'before and after' profiling. This may involve tidying up some of the test data used to test the passes. Study `latex` in anticipation of the progress report and dissertation. Time permitting, start implementing the project extensions if they were studied in the previous work block.

Deliverables: Some test programs which can be used for profiling, possibly in languages other than C++ which have an LLVM front-end (Haskell, Objective-C).

Lent Term

Weeks 1 and 2

Progress Report. Write up the progress report for the project so far, and prepare for the interview and presentation. Consider continuing work on project extensions.

Deliverables: Progress Report.

Weeks 3 and 4

Last chance to polish the implementation and work on project extensions. After this block, I will go into 'code lockdown', fixing only critical bugs. Research tools for profiling in anticipation of the evaluation. Begin planning the dissertation.

Deliverables: A working project! Plus any extensions that were managed. A high-level plan of the dissertation.

Weeks 5 and 6

Evaluation. Put together a final selection of code examples for project evaluation, which will be inserted into the dissertation. Collect all the profiling data required for a full evaluation of the project. Write pieces for each code sample explaining what transformations occurred and why, with the intention of inserting these pieces into the dissertation.

Deliverables: A comprehensive evaluation of the project, including all completed extensions, which should be of dissertation quality.

Weeks 7 and 8

Dissertation — first draft. Flesh out the details of the high-level dissertation plan written in Weeks 3 and 4. Write a first cut of the Introduction, Preparation and Implementation sections of the dissertation. Combine this with the Evaluation written in Weeks 5 and 6 to form the first draft. Present the first draft to supervisor for feedback.

Deliverables: A first draft of the dissertation, featuring four of the five key chapters.

Easter Holiday

Dissertation — second draft. Take on the feedback from supervisor and polish the Introduction, Preparation, Implementation and Evaluation. Write all missing pieces of the dissertation (Cover Sheet, Proforma, Table of Contents, Conclusion, Bibliography, Appendices and Index). Present second draft to supervisor and DoS for feedback.

Deliverables: A second draft of the dissertation, featuring *all* sections.

Easter Term

Weeks 1 and 2

Dissertation — final draft. Take on all feedback from supervisor and DoS. If not already done, edit down the dissertation to 12,000 words, moving code samples to the appendices as necessary. Proofread. Submit.

C.6 Resource Declaration

I will be using my laptop computer as part of this project, running a dual boot of Ubuntu and Windows 7. I can use the MCS as a backup.

For the evaluation, access to a highly multi-core machine would be useful to evaluate scalability. My laptop has two cores (doubled to four with hyper-threading), which should give useful results, but it would be nice if these results can be compared to that of a 32-core or higher machine, to see just how much parallelism can really be exploited.

The extensions feature distributed-systems parallelism, which would require some test machines. The SRG test cluster may be appropriate for this. Alternatively, virtual machines could be used, such as the Amazon EC2 VMs.