The University of York Department of Computer Science

Submitted in part fulfilment for the degree of MEng in Computer Systems and Software Engineering

A language to enable supervisor-level microthreading on the x86

James Molloy

10th May 2010

Supervisor: Dr C. Crispin-Bailey

Number of words = 21480, as counted by detex -e 'lstlisting,verbatim' | wc -w.
This includes the body of the report, the bibliography and all appendices.

Number of pages = 68, excluding any intentionally blank pages.

Modern computers are being built with more and more execution cores. Other papers and advances are changing our programming model from pure sequential to a more parallel friendly paradigm - many of these rely on many tiny threads (μthreads) executing concurrently. These implementations are inefficient as they are built on top of an antiquated untrusting kernel model, usually mapping many user threads to kernel threads.

This project investigates the speedups that could be gained by implementing μthreading at the OS level and below. In other words, if the OS and the language in which it is written were designed for modern multi- and many-core systems, what could be the benefit?

The project is based on an experimental bytecode language called "Horizon", developed between September and November 2009 by myself and Henry Harrington. The language is my sole work from September 2009 when this project began and Henry left.

Acknowledgements

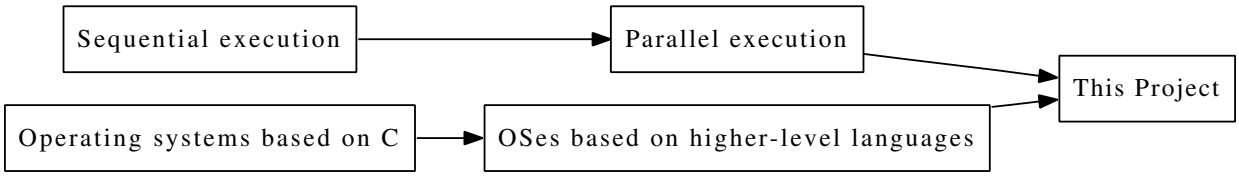
I would like to thank the following people for their input and advice that made my own life a lot easier.

- Henry Harrington, who had the inital concept of Horizon and wrote the initial Core implementation, as well as a start on the code generator.
- Alex Smith, who wrote and maintained the build system for Horizon.
- Dr Chris Crispin-Bailey, who supervised and mentored me through the project.
- The inhabitants of the IRC channel #osdev on freenode.net who helped proofread and sanity check the report.
- The LLVM developers at #llvm-dev on oftc.net, who helped me through some problems with their framework.

Contents

List of Figures

1. Introduction



Advances in computer architectures are currently creating systems with many compute cores available, so that multiple operations can be performed in parallel. Most programmers are still stuck in a somewhat sequential mindset however - partially this arises from the ubiquity until recently of the single core processor, partially from the logical, algorithmic way we solve problems. Other languages and paradigms are removing this barrier and allowing us to harness the power of parallel processing more easily.

Many of these languages require the use of many tiny threads called "microthreads" that can be scheduled across multiple processors. A problem is that with most current operating systems the only way to allow code to run on other processors is through the use of very heavyweight *kernel threads* - creating lots of these is slow and consumes resources. There are ways to combat the problem such as mapping lightweight user threads onto each kernel thread, but there is still a substantial difficulty and performance bottleneck.

Concurrently, advances in compiler technology and CPU power have brought bytecoded and just-in-time compiled languages to a comparable level of performance to that of native code. Using just in time compilation and compiling from an intermediate language that allows no unsafe constructs allows the use of *language-enforced isolation* - isolating one process or part of a process from another by the mere fact that the language does not let them interact.

This project aims to merge the two strands together and investigate how the use of a purpose built language that could trust the code it generates as the basis for an entire operating system could enable much more efficient use of parallelism. If the language were built with microthreaded parallelism in mind, speedups could be made in several ways:

- Moving the implementation of very lightweight threads down the OS stack to a sub-OS level means that they can bypass all other heavyweight constructs and be as quick and simple as possible.
- Because all code running on the processor is trusted (it has been compiled directly from a safe language) the OS can employ less hardware process isolation and speed up both context switching times and inter-process communication & I/O.
- With a dynamic recompiler at runtime, hotspot and profile-driven optimisations are possible, resulting in faster code than using a standard compiler.
- A safe language would use no pointer arithmetic, unlike C. Therefore pointer aliasing and other such assumptions that compilers have to make in order to produce valid code at the expense of optimisation are nullified, so the compiler can produce faster code.

This project details the research for, design of and implementation of such a language. Because the x86 is the most popular CPU in the desktop and server computing market, it will be tailored to that architecture. The project is laid out as follows:

Introduction This chapter provides an overview of the project.

Literature Review Investigation into prior research and the emergence of this field. Investigates the history of microthreading in both hardware and software, the history of operating systems that use bytecoded languages and some background on the compiler framework "LLVM", used in the project.

Problem Analysis Building on the literature review, this chapter provides an analysis of the current situation and a perspective on why this project is needed or useful. It concludes with the hypothesis that will be attempted to be shown true in the rest of the project, and a specification for the deliverable.

The Horizon language Introduces the "Horizon" language - an open-source language, revised and adapted for this project. This chapter details the design decisions that have taken place in its design, an overview of the language and how it is intended to be used, concepts and constructs, and a brief (2-page) syntax reference.

Context switch model Sidetracking a little, this chapter performs an investigation and experiment into the best methods of context switching for tiny threads. The context switching method is important to how the language is implemented and the code it generates.

Evolution of the parallelism model This chapter details the problems encountered when implementing the parallelism model in Horizon and the solutions found, culminating in the final model.

Implementation Details how the different components of the entire Horizon compiler were implemented at a lower level, module by module.

Evaluation Performs functional evaluation of the deliverable to the specification and performance evaluation versus both native code and with differing numbers of microthreads.

Further work Evaluates the potential for further work on the language, and what this work could include.

Conclusion Summarises the project and puts forward my conclusions and views.

2. Literature Review

The term "microthreading" (µthreading) has been used to describe several different yet related concepts in hardware and software.

2.1. µthreading in hardware

In hardware, the introduction of the term µthreading can be attributed to Chris Jesshope and Bing Luo in 1996 [28]. In 2000 Jesshope introduced "microthreading" as a method to reduce pipeline flushes and stalls in RISC architectures [28].

RISC CPUs are heavily pipelined in order to increase instruction throughput (instructions per cycle, IPC). The execution of instructions is split up into stages (such as instruction fetch, decode, ALU, writeback). While one instruction is being decoded, others are in their ALU stage, writeback stage, etc. This causes more of the CPU to be active at once. However, IPC is only increased to its optimum when the pipeline is full.

When the CPU decodes a nondeterministic instruction such as a conditional branch/jump, it is possible that the condition that determines whether to take the branch is still being calculated somewhere else in the pipeline. This causes a *pipeline stall* - the execution of instructions cannot be continued until the jump condition is calculated. The same stall is caused by "hazards" - reads after writes for example, where the content of a memory location may be modified by a previous instruction that is still executing.

2.1.1. Microthreading

Existing technologies such as super-pipelining and superscalar design, Jesshope argues, increase IPC in the best case but exacerbate the problem of pipeline stalls.

Attempts to reduce the impact of pipeline stalls include speculative instruction execution and branch prediction, however Jesshope argues that while these technologies do reduce the problem, there is a large price paid in both execution time when a mis-prediction is made and silicon area to deal with mis-prediction clean up.

Jesshope then gives a new solution - microthreading; [28, pg 2]

> The pipeline was kept full by allowing instructions to be executed from one of two possible sources. Where instruction execution was deterministic, the next instruction could be provided by the current thread in the normal manner. However, in situations where speculation would normally be required, either through control or data-dependency, it is not possible to execute the next instruction from the same thread without using speculation. In these circumstances the micro-threaded pipeline issues the next instruction from a new thread **without having to flush the pipeline**.

The above quote describes succinctly the operation of microthreads as Jesshope designed them.

Microthreads were designed to be very lightweight - they were invisioned to be drawn from the same context, so no context switching would be required to switch threads and a thread can be represented solely by a stack pointer / instruction pointer (SP/IP) pair. This has the disadvantage that compiler assistance is required; when splitting a task over multiple threads the compiler needs to ensure that those multiple threads do not use the same registers for their calculations, as other threads would destroy the values in them!

This then requires code similar to loop unrolling: if a loop is parallelised to X µthreads, X different versions of the loop are required, with the further restriction that;

$$n \times X \leq N \tag{2.1}$$

where n is the number of registers used by each µthread, and N is the total number of registers in the register file.

Analysis

Microthreads in hardware as Jesshope describes them seem an oversimplification: by simplifying the thread context to just a SP/IP pair, more work is forced on the compiler, causing several problems:

Scalability Because of equation 2.1 scalability is severely limited. Assuming a standard register file of around 32 general purpose registers [1], and assuming as in the example [28, pg 4] 5 registers per µthread, it only scales to a number as small as 6 (approx.) threads which can be drawn from the same context at one time.

Adaptability The compiler has to know statically which µthreads are running at any one time, in order to perform proper register allocation. Not only does this add extra complexity to the compiler, but it also requires that the available compute resources (register file size etc) are known at compile time. The program cannot dynamically adapt to increased processing power or an increased register file size - it requires a recompile.

Distributivity µthreading is specifically designed as an alternative for speculation, which implies a single physical core. In section 2.3 of [28] Jesshope describes how the algorithm can be adapted for multi-CPU systems. He suggests rolling the µthread bodies back up into one parametric body and dynamically allocating registers (on any CPU) on a per µthread basis. This not only alleviates the distributivity problem, but the adaptability problem above also.

2.1.2. Hyperthreading

In 2002 Intel released it's Foster-MP based Xeon processor [25] with "Hyper threading technology"™. Hyperthreading is Intel's extension of simultaneous multithreading (SMT) [1], which was introduced in a paper by Susan J. Eggers et al. in 1997 [21]. The essential idea behind SMT and HTT is the same as Jesshope's micro threading except that some processor resources are replicated for each running hardware thread, such as the register file (and optimisations such as register renaming files and call stack prediction registers) [1] instead of Jesshope's sole SP/IP pair.

SMT is used as an extension to (as opposed to instead of) the superscalar speculative execution model of Intel chips - when the speculative execution engine cannot predict where control flow would move to or the chip encounters some other wait condition (e.g. cache miss, TLB miss, HLT/PAUSE instruction), then the other thread is switched to.

There is an inefficiency in that every pipeline stage can only be filled by one of the two threads. If, for example, the out-of-order execution engine discovers that two instructions from thread 1 can be run on one cycle, they will be scheduled. With an issue width greater than 2, there are unused pieces of hardware - more instructions could have been scheduled to run. With pure SMT, the other thread would only ever be considered when *no* instructions from thread 1 are schedulable.

HTT extends this by attempting to fill empty pipeline slots from the other thread. Taking the above example, the first two slots in the first pipeline stage would be scheduled from thread 1, and as many others as possible (until the issue width is reached) would be scheduled from thread 2.

This means that the two threads can truly execute simultaneously, and the possibility of one thread starving the other of execution resources is reduced (but not eliminated).

Performance impact and controversy

HTT was criticised by embedded chip manufacturer ARM for being energy and cache inefficient in 2006 [23]. ARM state that it is considerably better to double your silicon area and add two cores than it is to opt for a more complex single core system with SMT support. ARM claim a 46% energy saving with a 4-core solution over a 4-thread single-core SMT solution [23].

Not only that but moving an application with two threads to a single SMT-enabled core will increase cache thrashing by 42%, whereas moving to two cores will decrease it by 37%.

Hyperthreading was removed for Intel's Core architecture but made a return in it's i7 architecture with two hyperthreads running on each of 4 physical cores.

[1] The MIPS architecture which Jesshope extends to add µthreading has 32 general purpose registers - not all of which are available for use by user programs (3 are reserved).

Analysis

HTT's approach is similar in concretion but different in abstraction from Jesshope's µthreading. The actual implementation (at least, in the more naive SMT form) is the same as Jesshope's with the exception of duplicated resources.

The way that the functionality is exposed to the user however is radically different. In HTT only two threads can possibly operate at once. There are no thread queues, unlike in Jesshope's system; the two threads are exposed at a higher level, each as an individual X86 core. To the user (in this case the operating system) there is no discernable difference between a hyperthread and a physical core. For optimisation purposes the OS *can* distinguish between physical and logical cores, but the usage is no different.

2.2. µthreading in software

In user software the term microthreading is not defined anywhere - instead it carries a meaning very similar to "protothreads", meaning a user-space threading library. There are a wide range of user-space threading alternatives, such as:

Kernel threads A kernel thread is a thread of execution as provided by kernel system calls. Kernel threads are the only type of threads (on UNIX-derivatives and Microsoft Windows systems) that can be scheduled to run on different logical CPUs. Kernel threads are pre-emptive in all modern general purpose OSes.

Green threads The term "Green thread" or "User thread" has been used to describe any threading mechanism that does not rely on the OS kernel and does not fall into any of the other categories. Green threads are less heavyweight than kernel threads, yet are not lightweight - startup and shutdown times are generally slow (but faster than kernel threads). An example of a green thread library is libpth (GNU Portable Thread library) [11] - this provides *pre-emptive* green threads for UNIX-like systems. Green threads can either be pre-emptive or cooperative.

Coroutines In 1997 Donald Knuth presented the idea of a "coroutine" [30]. Knuth presented coroutines as a superset of subroutines: a subroutine has one point of entry and multiple points of exit - a coroutine has multiple points of entry and exit. "Subroutines are special cases of ... coroutines" [30].

As coroutines can be suspended for any period of time, more than one program stack must be used. Some coroutine implementations use one stack per coroutine, some use continuation passing style [19]. Coroutines are all cooperatively scheduled (non-preemptive).

Protothreads Introduced in 2006 by Dunkels et al. [20], protothreads are tiny stackless threads. The stackless nature of the threads makes scaling to thousands of threads very manageable. Like coroutines, protothreads are cooperatively scheduled.

Fibers Fibers are cooperatively scheduled user-mode threads, solely available on Microsoft Windows operating systems.

The terms are very similar, so a rough tabular comparison is presented below;

	Preemptable?	Requires specific programming model?
Kernel threads	Yes	No
Green threads	Yes	No
Coroutines	No	Yes
Protothreads	No	No
Fibers	No	Yes

Recently there have been several projects developing "microthread" like infrastructures, or infrastructures aimed at delivering fine-grained control-flow parallelism.

2.2.1. Grand Central Dispatch

Grand Central Dispatch (GCD) is a technology released by Apple in September 2009, aiming to optimise application support for multicore computers [17]. GCD uses kernel threads as its low-level implementation

mechanism, but aims to abstract these away from the programmer, providing instead a set of functions that can run blocks [2] in parallel.

As Apple puts it:

> The dominant model for concurrent programming - threads and locks - is too difficult to be worth the effort for most applications. To write an efficient application for multicore using threads, you need to:
>
> - Break each logical task down into a single thread
> - Lock data that is in danger of being changed by two threads at once
> - Build a thread manager to run only as many threads as there are available cores
> - Hope that no other applications running on the system are using the processor cores

[17]

GCD's execution model and API revolves around 'blocks' and 'queues'. Blocks of code (using Apple's `^{...}` notation) can be put on a queue to run in parallel with other code. Each underlying kernel thread (it is assumed/made sure that each is operating on a different logical core), when idle, will be assigned a queue and will execute blocks queued on it sequentially. This way code can be scheduled across multiple cores, dynamically adapting to system load, yet the programmer is still able to deterministically specify that certain blocks must be run before other blocks.

GCD also allows blocks to be "parked" on event sources - the blocks are run whenever an event occurs.

Analysis

As extolled by Apple [17], GCD's benefits are fourfold:

Improved responsiveness Because small chunks of code can be moved off the main thread, "GCD helps developers make applications more responsive to user input".

Dynamic scaling Because GCD understands the current system state (on Mac OS X only) it is able to dynamically scale the number of threads in the thread pool depending on system usage.

Better processor utilisation Because GCD allows parallel programs to be written more easily, more cores on multicore systems are used more of the time.

Cleaner code Apple argue that GCD reduces the complexity of (user) code, "drastically improving readability, maintainability, and correctness".

Apple also claim that to queue a block on a queue requires just 15 instructions [17], which is extremely efficient.

The underlying implementation of GCD uses pools of kernel threads. Synchronisation between kernel threads uses heavyweight libraries such as `libpthread`, and although Apple have tried to use lock-free algorithms where possible, locking is still required at points.

The underlying mechanism has no scheduling. It is not pre-emptive, it is not even cooperative (depending on your definition) - the block runs until it is complete. This means that a block can, through maliciousness or buggy code, hang a thread indefinitely - seriously impacting system performance.

2.2.2. Thread Building Blocks

Intel's thread building blocks (TBB) are a similar solution to Grand Central Dispatch. Instead of augmenting the target language (C++, in TBB's case) with extra functionality as GCD does, TBB relies on the template metaprogramming techniques available with the C++ language and more specifically C++0x, the latest version of C++ which supports lambda functions as standard.

Similarly to GCD, TBB executes blocks of code (termed "tasks") one after another (with no timeslicing, preemptive or otherwise) [27].

[2] GCD augments C to add a closure-like '`^{...}`' notation for "blocks" of code, including arguments and data.

2.2.3. OpenMP

Open Multi-Processing (OpenMP) is an API that supports multiprocessing in the C, C++ and Fortran languages.

OpenMP is implemented as a compiler extension in several compilers including the GNU compiler collection (GCC), Intel Parallel Studio and Sun Studio (for Sun Solaris OS).

Semantically the interface is accessed using C preprocessor `#pragma` commands, specifying clauses such as scheduling, parallel FOR loops and synchronisation.

2.2.4. Stackless Python

Stackless Python (also called simply Stackless) is a flavour of the Python programming language that adds support for fine-grained parallelism. It uses microthreads (internally referred to as "tasklets") as its primary parallelism construct and can schedule cooperatively or preemptively.

Stackless has been used extensively in the implementation of "Eve Online" - a massively multiplayer online game - to provide for concurrency [29].

2.3. Bytecoded operating systems

Recently several groups have started producing operating systems based on bytecoded languages such as Java and C#.

2.3.1. Managed code and bytecode

The term "Managed Code" was invented by Microsoft to describe code that could only be run when supported by its Common Language Runtime (CLR). The more general term "bytecode-compiled" refers to other such languages too such as Java and OCaml.

Historically, computers were slow and compilers were inefficient and compute-intensive. This made compiling code extremely slow and coupled with the requirement of the many header files of different versions required by the ubiquitous C programming language, it made sense to compile once, then distribute.

Recently the speed of computers has increased to the point where compilation isn't the sluggish thing it once was (for certain languages, at least).

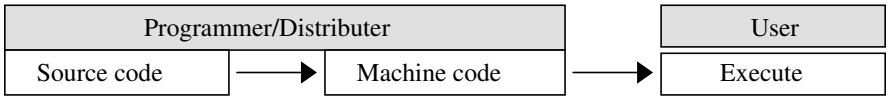


Figure 2.1.: Historic code compilation & distribution method

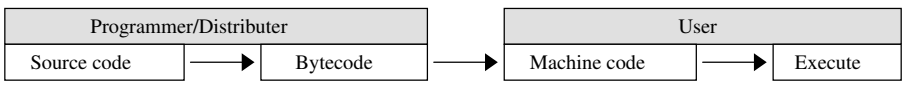


Figure 2.2.: Bytecoded code compilation & distribution method

The historic way to compile and distribute programs is illustrated in fig 2.1. Source code is compiled for the specific target operating system and machine to an executable binary file containing native instructions to be

executed on the processor. This file is transferred to the recipient, who executes it. The recipient's OS sets up an environment and causes the processor to jump to the location of the instruction stream in the file[3].

The newer bytecode-compiled method is illustrated in fig 2.2. Source code is compiled to an intermediate form (IR), which is normally a highly abstracted machine code-like architecture (also typically a stack-based architecture as in CIL and Java IR) - highly simplified from the original source which improves later compile times. This intermediate form is distributed to the recipient. The recipient cannot execute it directly but must launch either an interpreter, just-in-time (JIT) or ahead-of-time (AOT) compiler to convert the bytecode to executable machine code.

An interpreter will execute the instructions in the bytecode one by one, with little to no optimisation. Interpreters are extremely slow, but can serve a purpose as they can provide much more detailed debugging information owing to the lack of optimisation (c.f. `hugs` vs. `ghc` for Haskell).

An ahead-of-time compiler will attempt to compile all of the bytecode to machine code (including optional optimisation), then start executing. This is much like the historic compile-then-execute model.

A just-in-time compiler will compile bytecode to machine code, but only when it is required. It is a lazy execution model - only compile a function when it is called. Because of this a JITC'd executable will have a larger startup time than the equivalent AOT compiled executable. It will also have a more nondeterministic execution time depending on whether a function has been called before.

A bytecode compiler has several advantages with regard to optimisation than a conventional compiler. As the executable is compiled when it is (about to be) executed, many facets of the environment are known. As a trivial example, in a UNIX system the super-user has a process ID (PID) of zero. The compiler can know exactly what the current user ID is, and so any code that is conditional on the PID (such as whether the current user is the super-user or not) can be evaluated at compile time and only the correct case code compiled.

Just-in-time compilers have the added advantage that they can know not just what code is executed but *how and how often* it is executed. As the program executes, "hot" parts of the code can be monitored and recompiled with more optimisations[4].

2.3.2. Bytecoded operating systems

Historically, operating systems were written in a mix of assembler and C. Kernel developers needed to be able to fine tune their algorithms without being too decoupled from the low level aspects of the system - as C is such a primitive language (in terms of abstractions) it was the language of choice. Assembler is needed in any OS to provide the C code access to privileged instructions that the C language has no construct for (such as `sti` and `cli` in x86 - allowing and disabling interrupts), and also for bootstrapping the system into a state where C code can run (flat memory model, downwards-growing stack set up, etc).

Because speed was the main objective in writing kernels, C continues to be the dominant language to this day, although many hobby operating systems have started using C++ for its better abstractions (classes, for example) [34, 35]. However, with the current improvements in compiler technology, the time has passed when a proficient assembler programmer could out-perform a good compiler with optimisations enabled. Compiler back-ends now tend to have input from chip companies such as Intel and ARM, who know better than any which instructions are optimised in the chip and which aren't. Compiled code now out-performs most hand-written code.

Because device drivers and many applications were written in languages such as C which compile directly to native code, the kernel was not able to trust anything. User code (and sometimes driver code too) were run in hardware isolation, separated from other processes by separate address spaces and unable to use privileged instructions (such as `hlt`, `sti`, etc) enforced by hardware.

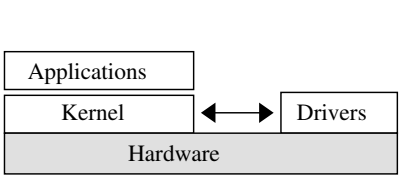
This hardware protection comes at a cost. Switching address spaces forces a flush of the translation lookaside buffers (TLBs) [26] which will then cause many memory accesses to walk the page tables again. Switching from user-mode to supervisor-mode costs 1348 cycles on a Pentium 4 [4].

Not only this but inter-process communication becomes more complex too as data either has to be copied from one program into the kernel (which is mapped into all address spaces) and back out into the target program, or a shared memory region must be set up.

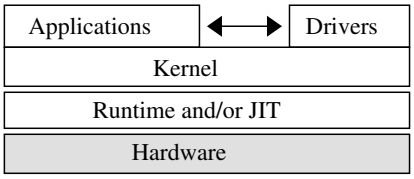
Both of these require significant overhead.

[3] This is extremely simplified, but serves for the purposes of this description / distinction.
[4] Optimisations increase compile time, so selective optimisation can result in both fast compilation and fast code.

Bytecoded operating systems are a redesign of the traditional kernel stack, illustrated in figures 2.3(a) and 2.3(b). The only native code run is the bytecoded language runtime and/or a just-in-time compiler (Singularity (see section 2.3.2) used an ahead-of-time solution instead). Everything else is written in a bytecoded language - including the kernel, all device drivers and all applications.



(a) Current / old style kernel stack



(b) Bytecoded kernel stack

Singularity

Microsoft Research has taken the step from using an unmanaged language such as C for the kernel to a bytecoded language - C#[5] - with their experimental OS "Singularity" [36].

Singularity was written from the perspective of dependability and trusted code. Microsoft wanted to be able to reason more about their kernel, and to remove the code repetition (lack of generics, large amounts of `void*`), type unsafety and manual memory management of C that causes many bugs.

Singularity is ahead-of-time compiled using the Bartok [36] compiler developed by Microsoft Research, and can run any CIL bytecoded program.

Because Singularity never runs native code[6] it can remove the use of many of the overhead-causing hardware protection mechanisms.

It uses a technique Microsoft calls Software Isolated Processes (SIP) - all processes are resident in the same address space but cannot interfere with each other as the bytecode in which they are written (compiled down to) does not allow unsafe pointer manipulation operations. Processes can only interact via interfaces exposed as OS "channels".

Singularity's channels take their roots from Hoare's Communicating Sequential Processes (CSP) [24]. Objects sent down a channel must obey a specific ordering and type protocol defined by a "Channel contract", an addition to Spec# in Sing#. This allows the OS to reason about and provide type safety for all applications.

Although the Singularity project was written with program robustness as the primary goal (see quotation 2.3.2), they ended up with a system that dominated all other modern operating systems in terms of general performance - See table 2.1. These performance figures demonstrate that bytecoded operating systems are in the very least competitive with current OSes.

Quotation 2.1 Singularity mission statement

"The Singularity project started in 2003 to re-examine the design decisions and increasingly obvious shortcomings of existing systems and software stacks. These shortcomings include: widespread security vulnerabilities; unexpected interactions among applications; failures caused by errant extensions, plug-ins, and drivers, and a perceived lack of robustness." [36]

Other bytecoded operating systems

Inferno Inferno is a bytecoded operating system initially produced in 1995, before Singularity. It is a "UNIX successor" based on "Plan 9 from Bell Labs", a research/hobby operating system originally created by the same group that developed UNIX and C [7] - Rob Pike, Ken Thompson et al.

[5] The language used is actually Sing#, a superset of Spec# which in itself is a superset of C#. Spec# adds contracts, Sing# adds contracts for channels.

[6] Apart from the fully-trusted native code that forms part of the kernel.

Table 2.1.: Performance of Singularity

	Cost (in CPU Cycles)			
	API Call	**Thread Yield**	**Message Ping/Pong**	**Process Creation**
Singularity	80	365	1,040	388,000
FreeBSD	878	911	13,300	1,030,000
Linux	437	906	5,800	719,000
Windows	627	753	6,340	5,380,000

It is a distributed OS with applications written in *Limbo*, a concurrent memory-managed bytecoded language. It is currently distributed with both Open Source and commercial licences by Vita Nuova Holdings.

JavaOS JavaOS was an experimental OS developed by Sun Microsystems as an OS with a Java virtual machine as a fundamental component [32]. It is primarily written in Java and has support for several architectures including IA-32, PowerPC and ARM.

JNode JNode (Java New Operating System Design Effort) is an open source project aiming to create a pure Java operating system. All of the kernel with the exception of some assembler stubs is written in ahead-of-time compiled Java. It currently supports TCP/IP networking and the ext2, FAT, NTFS and iso9660 filesystems [10].

JX Similar to JNode, JX is an open source Java operating system developed by the University of Erlangen. It uses an extended Java virtual machine to add support for hardware access and protection domains [22].

SharpOS SharpOS is an Open Source operating system written mostly in C# (with the .NET libraries). It is not yet production-ready and has only a basic command- line interface.

The OS has its own C# ahead-of-time compiler so that more of the system could be written in managed languages [14].

Cosmos Cosmos is very similar to SharpOS, being an OS written in C# and including its own ahead-of-time CIL bytecode compiler ("IL2CPU"). It only supports the x86 architecture and unlike SharpOS can only run code compiled with Microsoft's .NET Framework and not the Open Source alternative "Mono" [5].

2.4. Low Level Virtual Machine

The "Low Level Virtual Machine" (LLVM) is a compiler framework supporting both static and just-in-time compilation. It is Open Source, sponsored by Apple and has gained much momentum as an alternative to the GNU Compiler Collection (GCC) [13]. Originally written by Chris Lattner and Vikram Adve at the University of Illinois at Urbana-Champaign, it is now maintained by Apple (who now employ Lattner) and a team of hobbyist volunteers [13].

Unlike GCC (written in C) LLVM is written in C++ and is fully modular, with none of the "spaghetti code" that GCC is famed for [8]. LLVM acts on an intermediate bytecode language based around an infinite register machine (c.f. Java and CLR bytecode which use a stack machine model) internally called "bitcode".

LLVM bitcode, unlike Java, Python, Ruby and CLR is not safe; it allows arbitrary bitcasts from one type to another and pointer arithmetic. Typesafe languages can be built on top of it with a translation layer from some "safe" bytecode format to LLVM bitcode, but the bitcode itself is inherently unsafe. LLVM has backends for almost all major architectures including but not limited to IA-32, Intel®64, PowerPC, and ARM.

LLVM provides a C++ API to programmatically create bitcode (such as when compiling from some higher-level bytecode) and also a parser to create bitcode from a human readable assembly-like syntax, shown in example listing 2.1.

LLVM bitcode is based around an *infinite typed register machine*, meaning there are an infinite number of registers (prefixed with `%`) and each have a type associated with them. Compare this with the CLR and Java, which both work on an infinite stack machine [31].

LLVM bitcode is in SSA (Single Static Assignment) form. Once a register has been written to, it can never be written to again. This simplifies alias analysis and with discovering register liveness. It has instructions for stack allocation, memory loads and stores, integer and floating point arithmetic and comparison (`icmp`/`fcmp`), and flow control. It is strongly typed and supports basic types such as integers and floats and 5 derived types: arrays,

Listing 2.1: An LLVM example

```
  ; Returns the number of newlines found in a buffer.
define i32 @count_newlines(i8* %buffer, i32 %bufsize) {
  ; Local variables.
  %i = alloca i32
  %num_lines = alloca i32

  store i32 0, i32* %i           ; Initialize to zero.
  store i32 0, i32* %num_lines   ; Initialize to zero.
  br label %loop  ; can't just fall through to %loop

loop:
  %i_val = load i32* %i
  %is_done = icmp uge i32 %i_val, %bufsize
  br i1 %is_done, label %done, label %continue_loop
```

structs, vectors, pointers and functions. It is a vectorising compiler and supports SIMD (Single instruction, multiple data) and MIMD (Multiple instruction, multiple data) instructions as well as standard SISD (single instruction, single data) instructions.

3. Problem Analysis

Current general purpose operating systems take their roots in the 1960s and revolve around the C and C++ programming languages. C is simple, fast and popular enough that it will likely never die completely. C++ is widely used for its object-oriented capabilities coupled with the raw power and speed possible as with C.

Because C and C++ are completely unsafe and can be made to access any area of addressable memory, operating systems that cater for them must put in place copious amounts of protection against both malicious and accidental bugs so they do not interfere with the kernel or any other running process.

This generally involves:

Process isolation via hardware protection. Each process runs in its own address space so one can physically not address memory used by another process. Switching to another process requires a hardware address space switch and TLB flush.

Kernel interaction via system calls, special functions that switch the processor to supervisor mode and pass parameters to the kernel. The kernel executes the desired action (a syscall parameter selects the function to perform) and marshals the return values back to user space. This requires two mode switches as well as copious sanity checking of parameters.

Inter process communication via one or more of several mechanisms:

FIFOs / Channels In UNIX, these can either be *pipes* or *sockets*. Both are first-in-first-out channels dealing with raw bytes. Some form of protocol (in the case of sockets) can be superimposed on top, or not (pipes are more often used for transferring raw data between processes rather than control negotiation).

Shared Memory Memory cannot in general just be passed from one process to another. Because the OS relies on hardware protection and the x86 (and most other mainstream processors) cannot protect on a sub-page granularity (sub 4KB), any memory that is to be shared between processes must be allocated on a page boundary (which is not the case for general `malloc`ed memory).

Memory to be shared must be declared as such beforehand (using `mmap` and the `shm` interface). Even with this, a pointer to the memory cannot just be passed to the other process. A shared-memory `id` must be given to the process, which must then use the `shm` interface again to obtain a local pointer to the memory. This is non-trivial and involves overhead.

Remote Procedure Call RPC and message passing are related - RPC is synchronous and Message passing is the equivalent asynchronous mechanism.

RPC involves calling a vectored function similar to a system call. The caller is blocked while it waits for the other process to execute the requested function and return a result.

Message passing Message passing is generally asynchronous. A message will have an identifier, which is used to inform the recipient how to handle the content, and an optional data payload. As it is asynchronous the caller does not block. This also means that no return values can be returned to the caller. To facilitate this many message passing systems send an asynchronous "acknowledgement" message back to the caller when the message is handled (possibly with a status code).

While processors have been optimised to make these techniques fast, they are nowhere near as fast as local calls - this is exemplified by the linux kernel's `futex` [15] mechanism - a lock that operates as much as possible in userspace, only needing a system call when the lock is disputed.

This reliance on hardware protection and unsafe languages means that syscalls are required to create threads that can be properly scheduled, and these are slow. The current mainstream OS architecture is not amenable to large numbers of small threads running over many processors.

In contrast, bytecoded languages can be written in a way that allows speed but ensures data and type integrity, allowing multiple processes to run in the same address space - *language-based protection*.

Purists may argue that using a bytecoded language in the kernel would result in severe system slowdown and

latency. Indeed, given how voluble Linus Torvalds was regarding using the almost C-speed C++ language in the Linux kernel (quotation 3) the likelihood of this view is high. However this view was also taken back when UNIX was rewritten from pure assembler to C (with assembler) "... that something as complex as an operating system, which must deal with time-critical events, had to be written exclusively in assembly language." [37]. I personally believe the statement is as untrue now as it was untrue then.

Quotation 3.1 Linus Torvalds on C++ in the Linux kernel [12]

"In fact, in Linux we did try C++ once already, back in 1992. It sucks. Trust me - writing kernel code in C++ is a BLOODY STUPID IDEA.
The fact is, C++ compilers are not trustworthy. They were even worse in 1992, but some fundamental facts haven't changed:
1) the whole C++ exception handling thing is fundamentally broken. It's *especially* broken for kernels.
2) any compiler or language that likes to hide things like memory allocations behind your back just isn't a good choice for a kernel.
3) you can write object-oriented code (useful for filesystems etc) in C, *without* the crap that is C++."

Bytecoding, whether ahead-of- or just-in-time, does not imply slowness. Popular bytecoded languages at the moment tend to be slower than unsafe languages due to the effort such languages go to to guarantee safety. For example, in many such languages every array access requires a full bounds check which is expensive, as well as heavy reliance upon exception handling which is very slow and adds overhead.

I believe that a middle ground can be reached in which the bytecode gives approximately an equal amount of safety as current untrusted code + OS/hardware protection; that is that bugs or errors in a program can cause the abnormal termination of that process but *cannot* affect the integrity of any other process.

This compromise in providing *just enough safety* to remove the need for hardware protection should allow benchmark performance comparable with C.

3.1. Hypothesis

> *Basing an operating system on a language which allows for language-based process isolation can allow multicore microthreading to be implemented more efficiently than traditional OS models. Additionally, such a language does not imply slowness - the speed of such a language can approach that of C.*

To test this hypothesis I intend to build a language subject to the following requirements:

1. The language must be safe. No instruction in the language should allow undefined memory to be accessed. A special case is defined for NULL-pointer dereferences. These are allowed, and should terminate the program safely without any memory corruption. This special case is added for performance reasons.
2. The language must allow efficient microthreading.
3. For the deliverable the runtime must provide for microthreading on a single core. The ability to extend to multiple cores must however be addressed in the design. This restriction is in place solely to limit the project to a manageable scope.
4. The language should have a runtime speed approaching that of C++ for relevant benchmarks.
5. The language should support the most-used programming paradigms - it *must* support procedural and imperative and it *may* support object-orientation.

4. The Horizon language

4.1. Design decisions

The work done on this project is based on an experimental language platform codenamed "*Horizon*". An incomplete, partially developed prototype was developed by myself and another programmer, Henry Harrington, and released as Open Source software in 2009.

I redeveloped the language and compiler to suit this project better, retaining the Core module, which was used relatively unaltered. The existing code generator was used as the basis for the current CodeGen module but was rewritten significantly.

Several design decisions needed to be made early on in the project; some of these were made by both myself and Henry prior to the project start date (October 2009) and others were made by myself solely after this date. The decisions in subsections 4.1.1 through 4.1.7 were made together before October 2009; those in subsections 4.1.8 through 4.1.10 were made by myself only.

4.1.1. Paradigms to support

The imperative and procedural paradigms were "musts" in the functional requirements (section 3.1). The object-oriented paradigm was optional - as it is so ubiquitous we decided to support it natively in the bytecode, not force a higher-level language to implement it "C-style".

The functional paradigm has become very popular recently, with the introduction of lambda functions in many popular languages and the increasing use of Haskell. Functional constructs can be expressed in an imperative way easily - the Haskell compiler does this and uses microthreads too. As functional constructs can degrade to imperative constructs, we decided not to support them natively in the bytecode.

4.1.2. Programming model

We decided to create the language in the "intermediate representation" style - much like very high level assembler code. This has several advantages:

- The language can be used as a target for multiple high level languages.
- Parsing the language is extremely easy.
- Reasoning about the language is also easy.
- Mapping to LLVM bitcode (see later) is simplified.
- Lexical scoping and other such constructs can be removed, simplifying parsing and code generation further.

Our representation uses typed registers - all registers have a strong type associated with them. This creates our type safety.

4.1.3. Reflection

As the language is designed to be used as a target for higher-level languages, we felt that it was important to design how APIs may be shared between high level languages. We decided that classes, functions and other API-dependent constructs should be identifiable in the bytecode. This means that a HLL compiler could, for example, inspect the bytecode of a module to link with and provide access to the classes and functions within in its own syntax and format.

This is called compile-time reflection and is already supported with success in the Microsoft Common Language Runtime.

4.1.4. Class model

There are only two main ways of modelling class inheritance in statically compiled languages; Multiple inheritance and class-interface.

Multiple inheritance is C++'s model, and while successful it is difficult to implement well. It has numerous ambiguities such as the classic "diamond problem" in which a base class's member variables can be accessed via multiple paths through the inheritance graph. This can be alleviated in C++ with the use of virtual inheritance, but this brings its own problems.

Changing the inheritance graph from a DAG to a tree removes this problem, and the "class-interface" model (for want of a better phrase) does this by forcing a class to have only one parent. This removes a lot of power, so each class may implement any number of interfaces.

Interfaces are abstract classes - they have no member variables, only functions, and can be viewed as contracts. If a class C implements interface I then C will contain definitions for every function declared in I.

The latter model is used most frequently (in Java and C#, for example) and has fewer pitfalls - this is the model we decided to use.

4.1.5. Back end

The major difficulty with creating a new language is the back end and optimiser. The generation of IR, its optimisation then native code generation is extremely difficult. Enter LLVM. LLVM is a new compiler architecture (see section 2.4) built to serve as an efficient backend for multiple languages. It has a language- and machine-agnostic intermediate language (LLVM bitcode) that it uses both for program input and optimisation as well as code generators for all major architectures.

LLVM is actively developed and sponsored by Apple. It also has an API interface for defining bitcode programmatically, which makes it ideal as a back end to our language.

4.1.6. Higher-order (derived) types

Higher-order types in this context refer to complex constructs such as classes. The language has only three of these - classes, interfaces and variants (discriminated union types).

These are sufficient to emulate the entire range of C++ constructs:

Enumerations Handled as classes with constant integer members.

Unions Handled as variants - safer than C unions as the correct type is guaranteed to be extracted from the union.

Classes Handled as classes.

Structs Are identical to classes. In C++ the only difference between structs and classes is the default access type - structs default to `public`, classes to `private`.

Functions Handled in C++ as function pointers, and handled in this language as instances of a built-in interface, "Runnable".

4.1.7. Generics and polymorphism

In order to remove possible bloat and keep the bytecode understandable we decided to handle generic/templated types in the bytecode itself rather than forcing the high-level language compiler to deal with any generics it may use and emit multiple versions of code for type specialisations. This also allows generics declared in one module to be used by a module linking with it without explicitly instantiating all possible type specialisations it may require beforehand (as needs to be done with C++).

Generics use type inference at compile time to infer all types from instantiated parameter types.

4.1.8. Error handling

The "standard" way of handling errors is to use a `try {...} catch {...} finally {...}` construct - code in the `try` block is executed and when an exception occurs the code in the `catch` block is executed. The code in the `finally` block is executed regardless of errors, and usually contains cleanup code.

When implementing this in native code, there are two main schemes. One adds checking instructions to every call that could throw an exception. This adds extra instructions to the hot path and the cleanup code embedded in the function adds more. Many otherwise unnecessary branches get added to the program, which not only slows it down but makes the optimiser's job more difficult.

The second scheme is more generally used - on an exception, execution moves to an "unwind handler", which inspects the instruction pointer and stack at the point of exception and unwinds the stack, discovering the call graph. It then looks these instruction locations up in a precomputed table, in which it can find the location on the stack of variables that need destroying and the location of the nearest applicable `catch` statement.

The latter scheme is more used as it does not add any extra instructions to the hot path and so optimises the most used case first (the old adage "exceptions should be exceptional"). The stack unwinding however adds significant overhead and is tricky to implement in a threadsafe manner, the exception header tables cause bloat and the entire mechanism is nontrivial to implement.

I[1] decided not to implement explicit exception handling because of the reasons above, and leave communication of error conditions and change in control flow to the programmer, in line with the expected semantics of a systems programming language striving for speed (C++'s exception handling is nearly never used in systems programming).

Events in the language that would normally generate exceptions (inheritance downcasts to an invalid type, attempting to retrieve the wrong type value from a variant, out of memory, null pointer exception) cause the process to terminate abnormally (without any effect to other processes in the system).

4.1.9. Just-in-time vs. Ahead-of-time

There are advantages to each scheme - With JIT compilation more facts are known about the environment and the way the code is being run, which allows it to dynamically recompile and adapt. However, the startup costs are large as everything needs to be compiled first. When generics are instantiated, only the instantiations that are actually called/used need to be compiled, which reduces code bloat.

With AOT compilation all the compilation is done at one time, which speeds up the startup time of the executable. There is little way to adapt the executable for different environments and dynamic recompilation is not possible. All generics that are referred to in the code must be instantiated at compile time (a la C++) which can cause large code bloat.

Horizon aims to support both mechanisms, as each have advantages and disadvantages. The preferred method is to AOT compile the executable, then JIT adapt it at run time so the startup costs of JITting are alleviated but dynamic recompilation is still possible.

4.1.10. Threading semantics

Multithreading is a very important part of this language. Language support is required to make creation and destruction of microthreads as fast as they need to be.

With regards to a semantic and syntax for defining parallelism (specifically control-flow parallelism - data parallelism can be achieved with a vectorising compiler) several models were considered:

- The very successful Erlang programming language uses a `spawn` command to asynchronously start a function running in a different thread. It returns a handle with which the calling thread can wait on the spawned one or destroy it.
- Eiffel's concurrency model is based on a paper by Bertrand Meyer[33] and relies on an actor / coroutine model. No explicit process branching or parallelism is given in the language, it is all left up to the runtime.
- Compositional C++ [18] and Handel-C, a C derivative hardware description language both use the same

[1] This was a decision on which Henry's and my opinions were divided. We originally decided to implement exception handling, then when he left I reversed that decision for the reasons mentioned above.

model of `par` and `seq` blocks. Declaring a `par` block causes every code inside to be executed in parallel. `seq` blocks cause code to be run sequentially - for example inside `par` blocks so sets of sequential statements run in parallel. The `par` block does not complete until all statements inside it have completed.

We decided that our language and aims (minimal power removed from the programmer) were not compatible with Eiffel's approach, and decided on a hybrid model of Erlang's and Handel-C's.

Horizon's parallelism comes in the form of `par{...}` blocks, just as in Handel-C, however we also provide an analogous `par-async{...}` block that causes the contents to be launched asynchronously; the block completes immediately and execution continues. This is a parallel with Erlang's `spawn` command.

4.2. Overview

Horizon is built on top of LLVM and is designed to be a high-level assembly style language, type-safe and with an easy mapping to LLVM bitcode. It is intended to be future proof (not just a concept language for this project alone) so includes constructs for the *procedural*, *imperative*, and *object-oriented* paradigms.

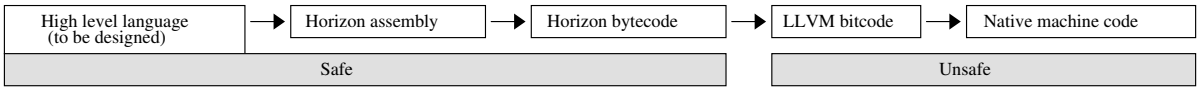


Figure 4.1.: Code transformation flow with Horizon

The functional flowchart can be seen in figure 4.1. A high level language - be it an existing language or one newly-designed - compiles down to Horizon assembly instead of its normal target (usually native machine code). That assembly is then parsed and converted to bytecode, which can then be converted to LLVM bitcode, with which LLVM can optimise and produce native assembly/machine code.

In order to allow multiple high level languages to be used in the same module (linking with packages created with a different language) and somewhat inspired by Microsoft's Common Intermediate Language, Horizon aims to be reflective in that it is descriptive enough to form a valid representation in any object oriented high level language (via the use of classes and interfaces). In this way Horizon can provide a language-agnostic description of an API and class structure.

Horizon is designed to work in ahead-of-time and just-in-time configurations. In the AOT configuration unsafe LLVM bitcode or native machine code is produced before the program is run, which can be a security risk. It is left to the hosting system and administrators to secure this code from malicious users.

Like LLVM horizon is based on an infinite register machine. Statements are of the vague assembly form `instruction dest-operand, src-operand` and in an attempt to prune useless instructions and only keep those with exact semantic meaning across all types, operations such as `add` which would normally appear as an instruction in high level assemblers are relegated to member functions of integer types - for example `x = a + b` in Horizon would be `call %x *%a.add %b`. As such, the list of instructions is short, and is listed in table 4.1. With hindsight the `trunc`, `sext` and `zext` instructions could be similarly removed as they only apply to integers.

4.3. General concepts

Horizon is based on an infinite register machine. The basic "variables" in a function are called "registers" and are prefixed with a percent (`%`) sign.

- Code can only appear inside function definitions.
- Data types can only be declared outside of function definitions.
- There is *no local scoping*, all registers in a function definition are in scope throughout that definition.
- Registers do not need to be declared before use.
- Registers are strongly typed.

Table 4.1.: Horizon instruction list

Instruction	Description
alloc	Object creation.
b	Unconditional branch to label.
bc	Conditional branch on an `i1` type to a true or false label.
call	Performs a function or member-function call.
dcast	Inheritance down-cast.
icast	Casts an object to an interface reference, assuming the object's class implements that interface.
load	Reads a global variable value into a register, loads a class member variable or performs array indexing.
mov	Copies the value of one register into another.
ret	Function return.
sext	Sign-extends an integer value.
store	Stores to a global variable, stores to a class member variable or stores to an array index.
trunc	Truncates an integer value.
ucast	Inheritance up-cast.
zext	Zero-extends an integer value.

- The type of a register is inferred from how it is defined - e.g. in "**mov** `%a, %b`" `%a` will take the type of `%b`. The power of type inference is in the allowing of OCaml style generics - see section 4.4.8.
- Register types cannot be redefined. Once their type is inferred it cannot be changed - a compile error will result.

4.4. Constructs

4.4.1. Types

Types in horizon can either be *basic* (integers, floats) or *derived* (classes, interfaces).

4.4.2. Basic types

The only basic types in Horizon are integers and floating point numbers. A floating point number can either be a **float** with a storage size of 32 bits or a **double** with a storage size of 64 bits. Both are stored in standard IEEE754 format.

Integers are more varied. There is a class for each bitwidth of integer from 1 to 64 named `i`n - **i1** .. **i64**. **i1** is a one-bit type - is Horizon's boolean equivalent and is used in conditional instructions.

There is also a **nativeint** type that takes the bitwidth of 32 or 64 depending on the executing processor. All other types including strings and arrays (characters are a special case and are of type `i20` as the full Unicode character set is 20 bits wide) are handled as classes and interfaces.

4.4.3. Global Variables

While frowned upon because of the reentrancy issues they can cause, global variables are allowed and use a syntax similar to the following;

```
variable myglobal : i8 = 5
```

The type is mandatory, the initialiser (`= 5`) is optional. If the initialiser is missing, basic types will initialise to zero and reference types will have an object created for them on the heap.

4.4.4. Classes

A class can contain member functions and variables. It can inherit from a single parent (single inheritance) and/or from multiple interfaces. This avoids the "diamond" problem of C++-style multiple inheritance in a standard way.

```
class MyClass extends MySuperclass
              implements MyInterface1, MyInterface2 {
  function (t) f ()
  function ()  g (%p : i8)

  variable myvar : i32 = 5
}
```

The listing above details the general syntax for declaring classes. We use the Java-style `extends` / `implements` notation for inheritance - it is well known, the meaning is obvious and does not rely on punctuation as the Ruby (`class A < B`) or the C++/C# (`class A : public B`) notations do.

Member function declarations are of exactly the same syntax as standard function declarations (see section 4.4.7) and member variables the same syntax as global variables (section 4.4.3). The optional variable initialiser serves as a default constructor in C++ parlance, ensuring that a valid object can be created without calling a constructor explicitly (an object is valid as soon as its `alloc` instruction completes).

4.4.5. Interfaces

Interfaces are abstract classes that only contain function declarations - they may not contain member variables. While a class can extend only one other class, it may implement any number of interfaces. This is standard practice in many object-oriented implementations (C# and Java to name two).

```
interface MyInterface {
  function (t) f ()
}
```

Interfaces cannot extend other classes or interfaces, and cannot implement other interfaces.

4.4.6. Variants

Variants are discriminated union types. These are types that contain one of several possible types, similar to a tagged union in C;

```
struct S {
  int ty;
  union {
    int i;
    double d;
    void *vp;
  } u;
};
```

In the above listing the union member `u` could either be an integer, a double or a void pointer. The struct member `ty` would hold a value between [0..2] indicating which of the members is valid.

This is unsafe: if `ty` is incorrect or buggy code causes the wrong union member to be read then invalid memory may be accessed (in all cases, but especially if the `vp` member is read when the union actually holds a `double` value!). Horizon's variant enforces that the correct union member is accessed in the language.

```
variant MyVariant {
  variable i  : nativeint
  variable d  : double
  variable vp : MyClass
}
```

4.4.7. Functions

Functions are the basic translation unit in Horizon. They are of the form **`function`** `(r) f(%p : t) {...}`, with this example declaring a function `f` taking one argument of type `t` called `%p` and returning one value of type `r`. The more general form for function definitions is;

```
function (return1, return2, ...) fn-name (%arg1 : type1, %arg2 : type2, ...) {
  ...
}
```

Note that functions can take any number of arguments and return any number of values. Class member functions also have an implicit `%this` parameter.

4.4.8. Generics

Horizon's generics take their roots and syntax from OCaml. They can exist in classes, interfaces and in functions; the types they take are determined at compile time. A generic is used and declared simply by prepending a quote (`'`) to an identifier, to create a unique generic type in that scope.

```
class MyClass {
  variable x : 'a
}
```

Similarly in a function definition:

```
function ('a) add_one (%arg : 'a) {
  call %x *%arg.add 1
  ret %x
}
```

The function above also shows a member function on a generic being called. By the time `%arg.add` is code-generated, the type of `'a` will be known, so the compiler can know exactly what member functions and variables the type `'a` has.

4.5. Instruction syntax

Instructions can only be declared inside functions. Each function has one main "block" of instructions, which must be terminated by a **`ret`** instruction. Labels are defined by an identifier postfixed with a colon (`mylabel:`).

4.5.1. alloc - Create an object

```
alloc %x : i32
```

Creates a new object of the given type (in this case `i32`) and stores a reference to it in the register (in this case `%x`).

The object is either created on the stack or on the heap. This is decided by the compiler and is determined by escape analysis on the register. If the object escapes to another function, it is created on the heap to be garbage collected. Otherwise it is safe to allocate the object on the stack.

4.5.2. b - Unconditional branch

```
b label
```

Unconditionally changes execution to the label specified.

4.5.3. bc - Conditional branch

(i) `bc %c, labeltrue`

(ii) `bc %c, labeltrue, labelfalse`

Depending on the value of the register (which has type `i1`), changes execution flow.

In the first case if `%c` is true, execution jumps to `labeltrue`. If it is false, execution flow does not change and the next instruction in the block is executed.

The second case differs in that if `%c` is false execution jumps to `labelfalse`.

4.5.4. call - Call a function

(i) `call fn 1, %a, 2`

(ii) `call %a, %b fn 1, 2, 3`

(iii) `call %a *%b.add 1`

(iv) `call %a *%b 1, 2, 3`

`call` has several syntax forms. In (i) a function `fn` is called with three parameters. `fn` here returns no variables.

In (ii) the same function is called but here it returns two parameters. The registers to store those return values in are written before the function name and are comma-separated.

In (iii) a member function `add` of register `%b` is called, with a single parameter `1` and a single return value, stored in `%a`. The asterisk before the register to dereference is required in order to distinguish return values from the call target.

in (iv) a "function pointer" is called. In Horizon "function pointers" are instantiations of the special interface `System.Runnable`, created with the **`mov`** instruction (section 4.5.8). Note the asterisk notation is still required in this form.

4.5.5. dcast - Inheritance down-cast

(i) `dcast %target : SubClass, %src`

(ii) `dcast %target : SubClass, %src fail faillabel`

The `dcast` instruction casts an instantiation of a class to one of its subclasses. It may fail if `%src` is not actually a subclass of `SubClass`. In this case, in (i) the program will be aborted with an error message, and in (ii) control will pass to the label `faillabel`.

4.5.6. icast - Interface cast

`icast %target : I, %src`

The `icast` instruction creates an interface reference from an instantiation of a class that implements that interface.

If `%src` belongs to class C which implements I, then `target` will have type I and reference `%src`.

4.5.7. load - Read/index variable

(i) `load %target, global_variable`

(ii) `load %target, %object.member`

(iii) `load %target, %array[%index]`

The `load` instruction has several forms. In its first form (i) it is used to load the value of a global variable into a register.

In its second form (ii), it is used to load the value of the member variable `member` from the object referenced by register `%object` into register `%target`.

In its third form (iii), it indexes an array. If `%array` is of class `System.Array` and `%index` is an integer or constant, `%target` is loaded with the value of the array at the specified index.

4.5.8. mov - Copy register

(i) `mov %dest, %src`

(ii) `mov %fnptr, func`

In the standard form (i), `mov` is used to copy the contents of register `%src` into register `%dest`. For derived types this means copying the pointer, *not* creating a new object in `%dest` and initialising to the same values as `%src`.

In the form (ii), `mov` is used to save a function reference into a register. The function `func` is transformed into an instantiation of builtin type `System.Runnable` and stored in `%fnptr`.

4.5.9. ret - Function return

(i) **`ret`**

(ii) **`ret`** `%x, %y`

Returns from a function. Can take zero or more arguments as values to return.

4.5.10. sext - Sign-extend integer value

`sext` `%a : i32, %b`

Sign-extends the value in `%b` to (in this case) `i32`, and stores the result in `%a`.

4.5.11. store - Write/index variable

(i) **`store`** `global_variable, %src`

(ii) **`store`** `%object.member, %src`

(iii) **`store`** `%array[%index], %src`

This is the opposite of `load` - see section 4.5.7.

4.5.12. trunc - Truncate integer value

`trunc` `%a : i8, %b`

Truncates the value in `%b` to (in this case) `i8`, and stores the result in `%a`.

4.5.13. ucast - Inheritance up-cast

`ucast` `%dest : Superclass, %src`

Casts `%src` to type `Superclass`, assuming that `Superclass` is a superclass of `%src`. Stores the result in `%dest`.

4.5.14. zext - Zero-extend integer value

`zext` `%dest : i32, %src`

Performs zero-extension in the same way that `sext` performs sign-extension - see section 4.5.10.

5. Context switch model

The threading model used - preemptive or cooperative - defines and shapes the rest of the language. It therefore must be decided on at an early stage.

Normally this would be an easy decision. Because most frameworks operate in userland under a *NIX kernel they only have the choice of "slow preemptive" or "fast cooperative".

We however do not have that limitation. Preemptive threading is slow in *NIX applications for two reasons:

1. If solely kernel threads are used, using the kernel to preempt them, setup and teardown costs are huge. The overheads are far too large for a microthreaded application.
2. If preempting solely in userspace (which is possible) then the timer signals that cause the preempt are UNIX signals. And are thus slow and have overhead. Doing this well is also extremely difficult owing to the lack of things one can actually legally *do* in a signal handler - the POSIX specification is extremely restrictive in what is guaranteed to run successfully.

Neither of these reasons apply to us; we define the exact overheads associated with our threads and we have direct control over the system timer signal (from either the PIC or APIC) to create minimal overhead.

Nevertheless, preemptive code has its problems. It suffers from a *lack of knowledge* about the subject program. For example if a program uses an SSE register or floating point register once, then context switches in future must save *all* SSE and floating point registers, regardless if they are being used as it has no idea which registers are live at any point. A cooperative system has that knowledge.

Because our system runs only trusted code we have more options for thread switching. Here I will evaluate four such methods; Standard preemptive, "partially preemptive", Naïve cooperative and cooperative with rate management.

5.1. "Safe points" and cooperation

Several of the switching methods explained below rely on at least some kind of cooperation between threads. This means that at several points in the program some code, C, must be added.

Where C is added is important. It must be added in all places such that a thread can never block or go into an infinite loop without encountering at least one instance of C. It must also be added in all places such that the time between invocations of C is not greater than some arbitrary bound. This is to try and ensure fairness between threads and stop starvation.

This may sounds like attempting to solve the halting problem and indeed it is similar, however in the halting problem a program is analysed and an answer is required as to whether the program *will* block when run with a certain input. We only need the answer as to whether the program *can* or *may* block when run regardless of input, which is a much simpler problem.

> Assume that no function call can block. Then, in a function F, there are only two constructs that change the flow of execution: branches (`b` and `bc`) and calls. We know that calls cannot block because of our assumption. Only loops formed from branches can then block. We can stop this by inserting C before every `b` and `bc` instruction.
>
> Then, we can insert C somewhere in F; in that way F cannot block, and our inductive assumption is maintained.

This isn't quite optimal however. Firstly, where should we position C inside F? We could put it at the start, somewhere in the middle or at the end, before the function epilogue. At the start of the function there will be live registers holding parameters. In the middle of the function there will be live registers holding partial results. However at the end of the function the only register that is likely to be live is that used to return the function result to the caller (in x86's case, `eax`).

Listing 5.1: Example partially preemptive switching code

```
  ...
  cmp byte [signal], 0
  jz  .after

  push eax ; Push all live registers, in this case just eax.
  call thread_switch
  pop eax  ; Pop all live registers back

.after:
```

The end of the function is therefore the best place to insert C, as it will not have as many live registers to save.

Secondly, Inserting C before *every* branch is excessive; it only really needs to be inserted before any branch that causes a loop. This is simple: to cause a loop there must be a *backwards* branch - a branch to a label before the instruction. So instead of inserting C before every branch, just insert it before any backwards branch.

5.2. Proposed algorithms

5.2.1. Preemptive

This standard method of thread switching assumes nothing about the code running; it does not require trusted code. On a timer interrupt it saves all registers (including FPU and SSE registers if required) to the call stack, switches stacks to a different thread, restores that thread's registers and returns.

The concept is extremely simple, however the lack of knowledge about the target thread means that all registers must be saved all the time, no optimisations can be made (such as not saving registers that aren't live).

5.2.2. Partially-preemptive

"Partially-preemptive" is a term used for lack of a better - this algorithm relies on the code being trusted and partitionable into non-blocking chunks. See section 5.1 for a full explanation of how this is done.

This partitioning can allow us to insert task switching code at the end of each block. When a timer interrupt arrives, instead of immediately and forcibly switching the running thread we set a global variable, `signal`, to 1.

At the end of every block we check the value of `signal`. If it is nonzero then all live registers are pushed to the stack and the thread switching function called cooperatively. On return from the function (which happens when the thread is rescheduled) we pop those same registers off the stack and continue.

This is demonstrated in listing 5.1. In this case the only live register is `eax`.

This approach increases the code size and adds instructions (including a nondeterministic instruction, `jz`) to the hot path of every block, potentially many times per function. Possibly offsetting this is the information that is known about the live register set at the end of a block.

5.2.3. Naïve cooperative

The most naïve of all the algorithms; just as above code is inserted throughout the program so that it cannot block, however this code *always* executes.

This obviously has the sideeffect that the switches are so fast that any overheads will be amplified, and registers cannot remain live for long before they must be saved.

However, it does remove a lot of complexity and is included to see if the simplest solution is the best.

5.2.4. Cooperative with rate management

This attempts to solve the problem of naïve cooperative task switching too quickly to get useful work done by attempting to manage the rate of context switches. Consider the code in listing 5.1(a) that defines the `thread_switch` function that is called at the end of every block, as in naïve cooperative, by the code in listing 5.1(b).

```
1 thread_switch:
  switch:
3         cli
          mov eax, [ticks_since_last]
5         mov dword [ticks_since_last], 0
          sti
7
          cmp eax, 1
9         jl  .too_low
          jg  .too_high
11        jmp .just_right

13 .too_low:
          mov eax, [n]
15        add eax, 10
          mov [n], eax
17        jmp .just_right
  .too_high:
19        mov eax, [n]
          sub eax, 10
21        mov [n], eax

23 .just_right:
          mov dword [m], 0
25
          ; Do the actual task switch here.
27
          ret
```

(a) Thread switch function

```
          push eax
2         mov eax, [m]
          inc eax
4         mov [m], eax

6         cmp [n], eax
          pop eax
8         jle .after

10        push ecx      ; Push all live registers -
                        ; ecx in this case.
12        call thread_switch
          pop ecx       ; Pop live registers again.
14 after:
```

(b) End-of-block code

Figure 5.1.: Cooperative with rate management

As a quick summary the code increments the global variable `m` until it equals `n`, at which point it does a task switch. Before it does the task switch it performs some analysis as to how quickly it is task switching with respect to the timer ticks. It adjusts `n` to attempt to switch at a rate approximately matching the timer interrupt rate.

The repeated code in listing 5.1(b) saves the scratch register `eax` on line 1, then increments the value in global variable `m` and compares it with `n`. If they are equal then all live registers are saved and the `thread_switch` function called.

The `thread_switch` function loads then resets the variable `ticks_since_last` which counts up on every timer interrupt. `n` is then adjusted depending on whether `ticks_since_last` was greater than, less than or equal to the target ticks-per-switch of 1. Finally `m` is reset to 0 and the actual task switch takes place.

This adds extra comparisons and accounting to the end of block code that will cause bloat, but should reduce the number of context switches drastically.

5.3. Tests

These tests were performed before Horizon was complete enough to use. As a consequence the tests were written in pure assembler or compiled to LLVM bitcode and then edited to insert the correct cooperative code. The same effect can easily be created with an LLVM-based compiler.

All tests had the same format; They were run on a bare-metal operating system I designed[9] with no other threads or any operation running in the background to skew the results. The system timer was set to differing frequencies, and each test was run for 2 seconds. For each frequency 5 or 10 readings (depending on the range of frequencies tested; test 1 used 5, every other used 10) were used.

On the graphs below it looks like only one point is taken for each frequency - they are actually clusters of 5

Listing 5.2: Fibonacci function

```
work:   sti
        mov eax, 1
        mov ebx, 1
        mov ecx, 0

loop:   call fib
        inc ecx
        jmp loop

fib:
        add eax, ebx
        xchg eax, ebx
        ; ... repeated x8 ...
        add eax, ebx
        xchg eax, ebx
        ret
```

or 10 points, just so close together that they cannot be distinguished.

5.3.1. Test 1: Fibonacci sequence

The first test involved a hand-crafted assembler function that computed the 10th and 11th fibonacci numbers. It can be seen in listing 5.2 - note that due to the repetition involved in the `fib` routine I have truncated the listing slightly. The register `ecx` tracks the number of times `fib` has been called since the `work` function was called (and `ecx` reset).

For the three methods that require a cooperative stub this was inserted on between lines 15 and 16.

Results

The results are shown in figure 5.2. For this test the partially preemptive method is by far the best. All methods except naïve cooperative are affected adversely by a high clock rate; this is to be expected as it multiplies all overheads.

Naïve cooperative performs well, however arguably this is its ideal test - a monolithic function without branches or loops.

All tests apart from cooperative are affected by a "laddering" for want of a better word at high ($>$ 30000 Hz) frequencies - remember that all points plotted are actually clusters of 5 independently obtained points, so these patterns are not anomalous, they are repeatable. It is likely that they have to do with timer inaccuracy at high resolution (the system timer on general purpose OSes is normally never run higher than 1KHz, here we go up to 100KHz.

5.3.2. Test 2: Dhrystone

Dhrystone is a CPU benchmark developed by Reinhold Weicker in 1984 [39]. Its name is a pun on Whetstone, a similar benchmark that also profiles floating point performance (Dhrystone includes no floating point operations).

The C implementation used was from [2]. It was compiled with Clang, the C language frontend for LLVM to produce LLVM bitcode. A Perl script was run on the resulting bitcode (in human readable assembler format) to insert the cooperative stubs (hereon after called "annotations") where needed. The script to do this is in listing 5.3. It inserts macro calls for the C preprocessor which is then run on the bitcode to produce the annotations properly. This was done so that "annotate.pl" could be run once and the result used with all different algorithms easily (by changing the `ANNOTATE()` macro definition).

Clang makes the job easier by naming all labels to do with loops "while*" or "for*". In order to find backward branches the script merely has to find branches to these labels, and the keyword `ret` for the end of the function.

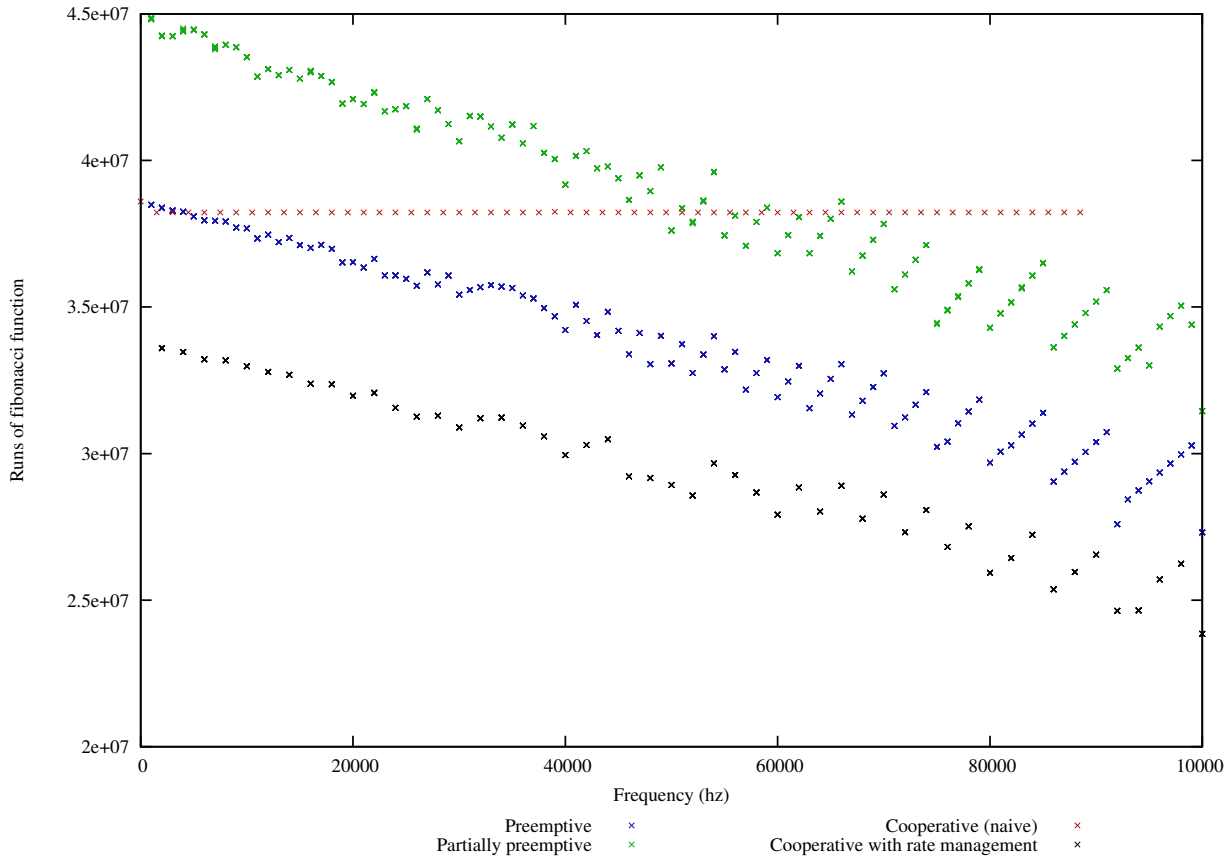


Figure 5.2.: Fibonacci results

Listing 5.3: annotate.pl

```
#!/usr/bin/perl

use strict;
use warnings;

print "#include␣\"annotation.ll\"\n\n";
print "TOP\n\n";

my $n = 0;
while(<>) {
    if(m/^( *)br label %(while|for)\.cond.*/) {
        print "␣␣ANNOTATE($n)\n";
        $n++;
    } elsif(m/( *)ret /) {
        print "␣␣ANNOTATE($n)\n";
        $n++;
    }
    print $_;
}
```

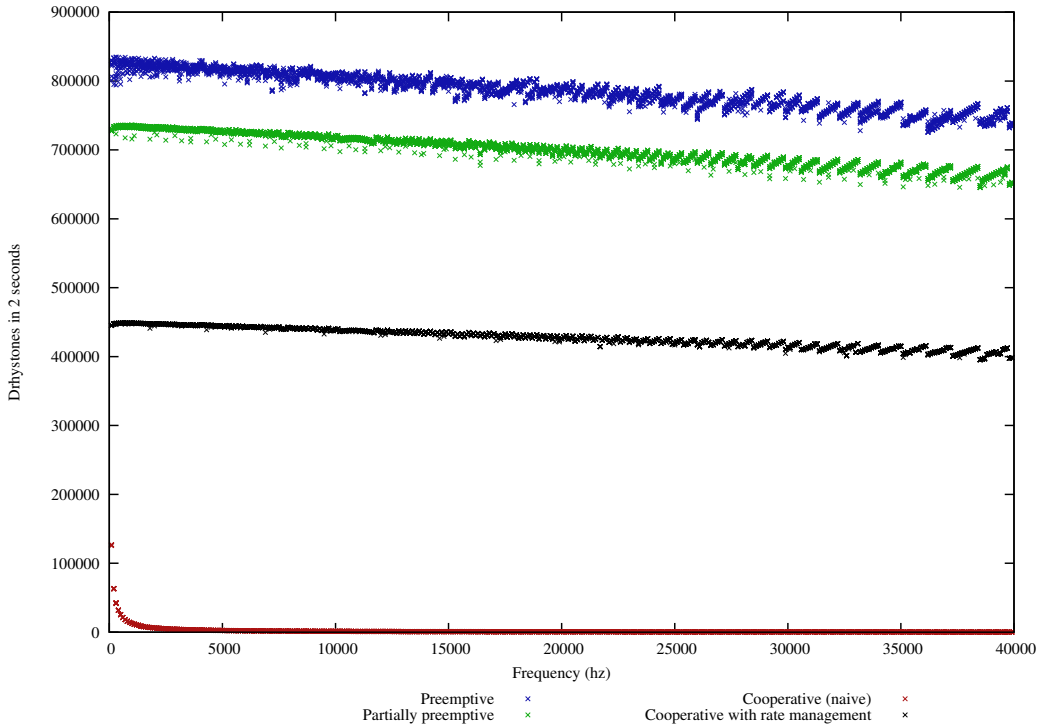


Figure 5.3.: Dhrystone results

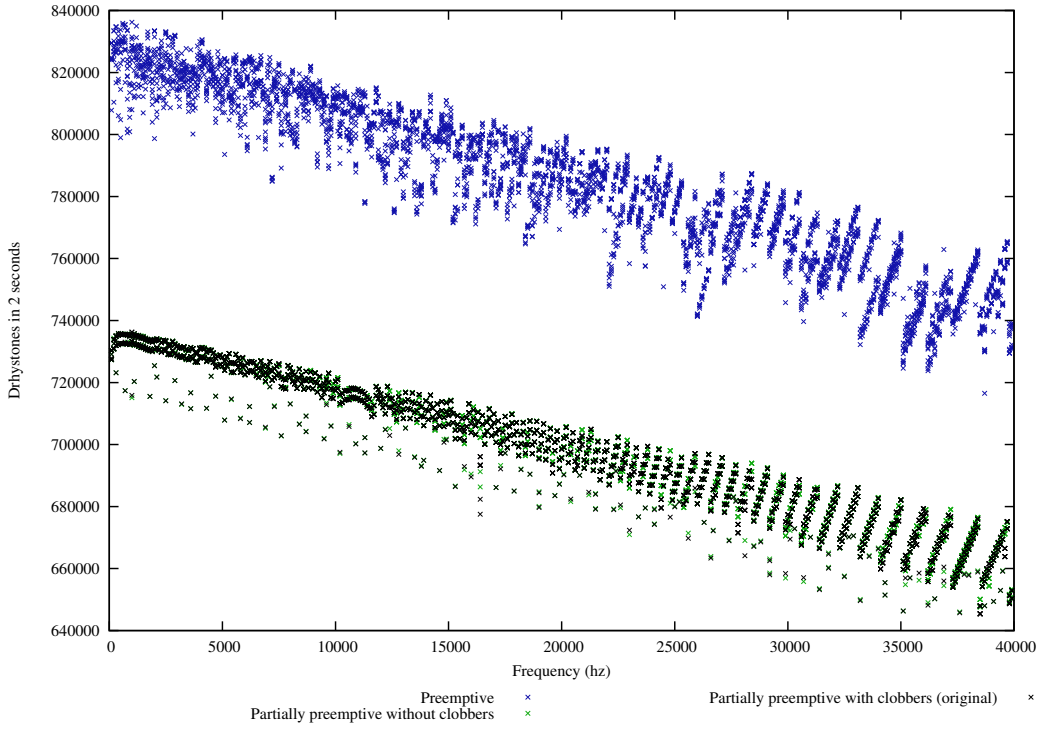


Figure 5.4.: Dhrystone results when removing caller-clobbers

Results

Analysing the results in figure 5.3 we can see that this time the preemptive method is by far the best, with the partially preemptive method slightly behind. Naïve cooperative is so slow that on the scale it appears to be almost zero - the improvement as the frequency tends to zero is likely indicative of how much overhead is taking place.

Cooperative with rate management fares better, being about 50% the value of preemptive, however is poor in comparison. Also note that the frequency domain tested has changed to [100..40000) in steps of 100Hz instead of the 1000Hz in test 1.

5.3.3. Test 3: Improving partially preemptive performance

The closest competitor to preemptive task switching in a real-world benchmark seems to be the partially preemptive method. There are no techniques available to speed up preemptive task switching, but there are to speed up partially-preemptive.

We shall attempt to improve its performance here.

Removing callee-save clobber

In the previous tests the annotation for partially-preemptive has been an LLVM inline assembler directive:

```
call void asm sideeffect "cmpb $$0, signal; jz 1f; call project_task_switch; 1:", "~{memory},
~{ax},~{cx},~{dx},~{bx},~{si},~{di},~{r8},~{r9},~{r10},~{r11},~{r12},~{r13},~{r14},~{r15},
~{dirflag},~{fpsr},~{flags}"() nounwind
```

That is, all general purpose registers get clobbered by the inline assembler call. This forces LLVM to save any it wishes to keep live over the call.

There are several registers - `ebx`, `edi` and `esi` for example - that are marked as "callee-save" by the C ABI. These registers can be assumed by the caller of a function to not be changed when the function returns.

As this statement informs LLVM that *all* registers are clobbered, it must then insert code to save the callee-save registers' value for its calling function. We may be able to improve performance by removing these clobbers.

The results can be seen in figure 5.4. Note that as the cooperative and cooperative-with-rate-management graphs have been removed, the scale has changed and that is why the points look less coherent.

The black points are with the original clobber list, and the green points are with the new clobber list. It can be seen that the changes have had no effect.

Changing jump order

The current annotation code performs a jump on the hot code path. When `signal` is not set (the common case), a jump takes place. This might be stopping the processor's speculation engine from performing efficiently.

The statement can be changed to;

```
  mov eax, 1f    ; Store the address of label 1:
  cmpb [signal], 0
  jnz do_task_switch
1:

do_task_switch:
  push eax
  jmp task_switch
```

Firstly the address to be returned to after a task switch should there be one is loaded into a scratch register. If `signal` is nonzero, control moves to `do_task_switch` which emulates a `call` by pushing the intended return address (in `eax`) onto the stack then jumping to the standard `task_switch` function. This emulates a conditional call, which the x86 does not have as an instruction, and causes the jump to only be taken in the uncommon case.

We could also pin the global variable `signal` to a register. Instead of performing a memory/cache access to get its value in every annotation, if it were pinned to a register (for example `ebp`) all that would be required is

a simple register comparison which may be quicker.

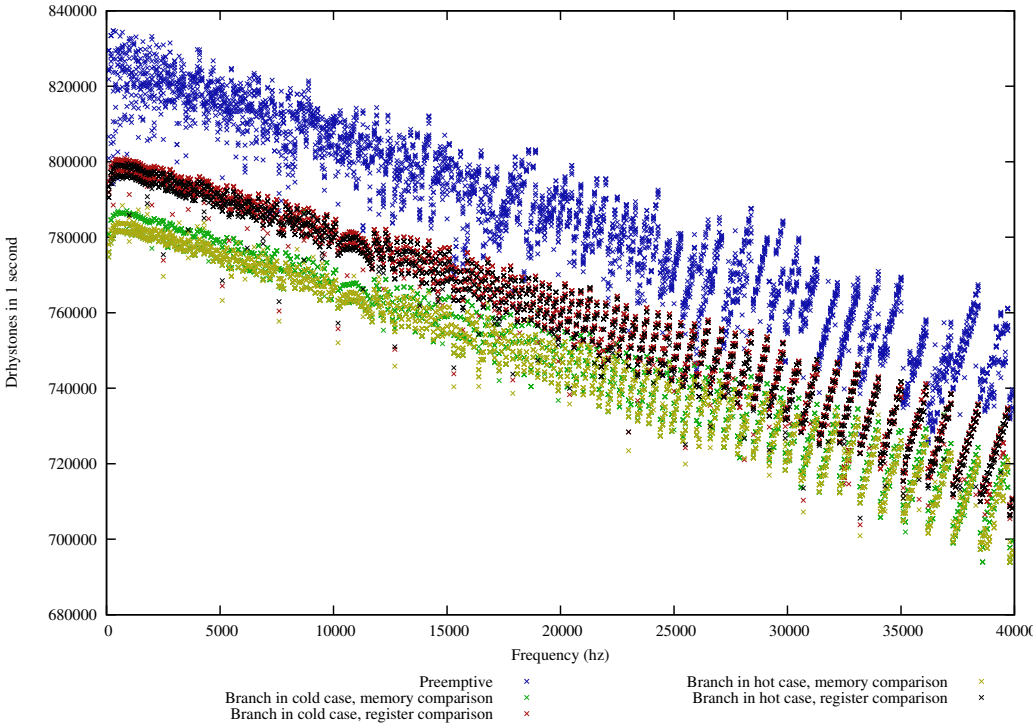


Figure 5.5.: Dhrystone results with hot/cold jumps and register/memory accesses

In figure 5.5 the results, while slightly messy, are conclusive. The red and black points consistently dominate the green and yellow points - this shows that the register comparison is much better than the memory comparison. The red points are also always better than the black points - this shows that the branch in the cold case indeed provides an improvement over the original hot-case branch.

It also shows that even though performance has improved, the partially-preemptive method is still dominated by the standard preemptive.

5.3.4. Test 4: Whetstone

The results so far have shown preemptive to be by far the best. This could however be because only integer arithmetic has been used so far.

The preemptive method has to save and restore *all* registers that could have been used. In integer arithmetic there are only 8 general purpose registers (16 on 64-bit x86) to save, so the savings the other methods can gain by knowing which registers are live is tiny.

In contrast, the floating point unit of the x86 provides another 10 registers to save (8 for the FPU stack, one status and one control). If floating point arithmetic were used, perhaps the gap between preemptive and partially-preemptive may narrow.

The x86 also has "Streaming SIMD Extensions", or SSE, instructions. These perform vector calculations and can be used for floating point arithmetic also - often faster than the raw FPU. They have their own registers (`XMMn`) and if SSE is enabled then preemptive context switching must perform a specific instruction to save the FPU and SSE state: `fxsave` (and its counterpart `fxrestor`). These instructions have to store and load *512 bytes* of register state to memory, more than standard floating point arithmetic and much more than just the general purpose registers.

Whetstone was alluded to in section 5.3.2 - it was the precursor to Dhrystone and includes floating point arithmetic. This will be run with both methods and with SSE enabled and disabled.

These results (figure 5.6) are more interesting. With SSE disabled, preemptive outperforms partially-preemptive massively. However when SSE is enabled preemptive is still better than partially-preemptive, but by a lesser margin.

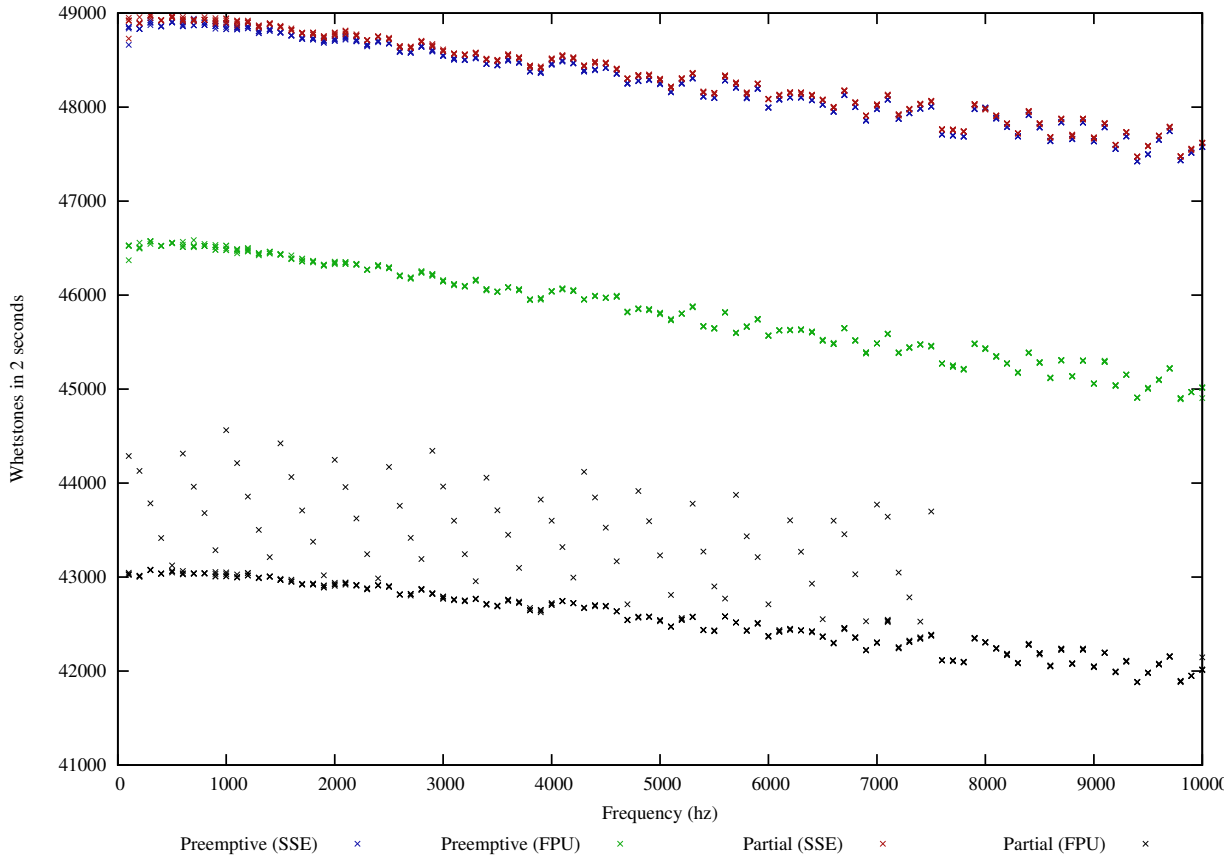


Figure 5.6.: Whetstone results with SSE enabled and disabled

I believe this is because when SSE is enabled the benchmark performs much better, reducing the effect of costs due to overhead as the thread switch is required fewer times per loop.

5.4. Conclusion

The standard preemptive algorithm works better than any other, in all situations. The partially-preemptive algorithm, its closest competitor, closed the performance gap to a very small amount in the SSE-enabled Whetstone test, but still did not surpass it.

Considering the SSE-enabled Whetstone was designed to be the worst-case scenario for the preemptive method and it was still more efficient, the preemptive algorithm is the best all around and should be used in Horizon.

This means that no cooperative annotations are required in Horizon's generated code, which actually reduces compiler complexity by a large factor.

6. Evolution of the parallelism model

6.1. Initial model

As discussed previously in "Threading semantics" in section 4.1.10 the parallelism model would be in `par/seq` style. I also decided to simplify the model for ease of compilation - the idea of `seq` blocks was removed and each `par` block was composed of multiple subblocks, each executed in parallel. For example:

```
call func1
par {
        mov %x, %y.member
        call func2 %x
} {
        call func3
}
```

In the above code snippet there are multiple braced regions. Code inside these regions execute sequentially, and the regions themselves are executed in parallel with each other.

6.1.1. Attempted implementation

The main problem with implementation was how to interface multiple threads with LLVM's sequential execution model. We allow arbitrary horizon code to be executed within a `par` block, and this code must have access to all registers declared in the current function scope. The par blocks must then be implemented in LLVM as subblocks of the containing function.

This creates a problem as we cannot pass the address of such a subblock outside of the function. LLVM stipulates that only functions themselves can have their addresses taken - internal labels cannot, for optimisation purposes. It is however necessary to interface with external code to spawn the microthreads - this involves some assembler-level trickery that LLVM does not allow.

I therefore had to find a way to jump to the correct label from *inside* the function, using some dynamic dispatch system.

The initial implementation was based on a UNIX-style `fork()/wait()` architecture and is detailed in mapping 6.1. The par block starts with a call to a runtime "fork" function. This takes as a parameter the number of microthreads required and will return a thread ID in the range $[0..n)$. This ID can be used to identify which thread is currently executing. This can then be fed to an LLVM `switch` construct that behaves the same as in C, to jump to the block that will perform the thread's work.

When the work is complete all threads jump to the same common "sieve" - `ParEnd1` in this example. At this point the threads are rejoined together. A register, `ParCounter0`, is initialised to the number of microthreads created. This can be seen in lines 1-2 of mapping 6.1. At wait time this register is atomically decremented (atomicity is very important to eliminate race conditions) and the resulting value passed to the `wait` function. The `wait` function simply kills the current thread if its parameter is not zero and returns otherwise. Thus in a "last through the gate shuts the door" approach all threads except the last to reach the wait statement are killed. Execution then continues sequentially.

This was implemented into Horizon and worked - the control flow graph (CFG) for the sample program snippet in listing 6.1 that computes if two numbers are prime in parallel is shown in figure 6.1. Note that in the CFG the last three lines of listing 6.1 have been removed to save space.

6.1.2. Problem - Phi functions and register allocation

The approach did have a large problem. It ignored an assumption that LLVM makes - only one path through a program can ever be taken at once. This assumption is used when performing phi transformations.

Listing 6.1: Parallelism test program

```
call %a *%num.add 1
par {
    call %y is_prime %a
} {
    call %x is_prime %num
}
call %r *%y.shl 4
call %r *%r.add %x
ret %r
```

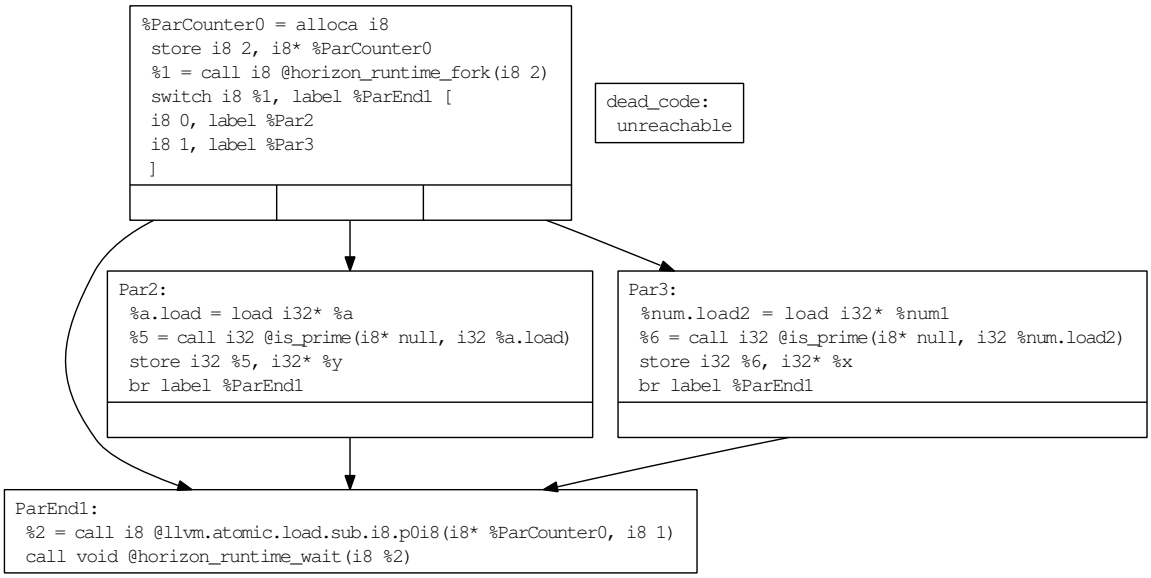


Figure 6.1.: Sample program with optimisations disabled.

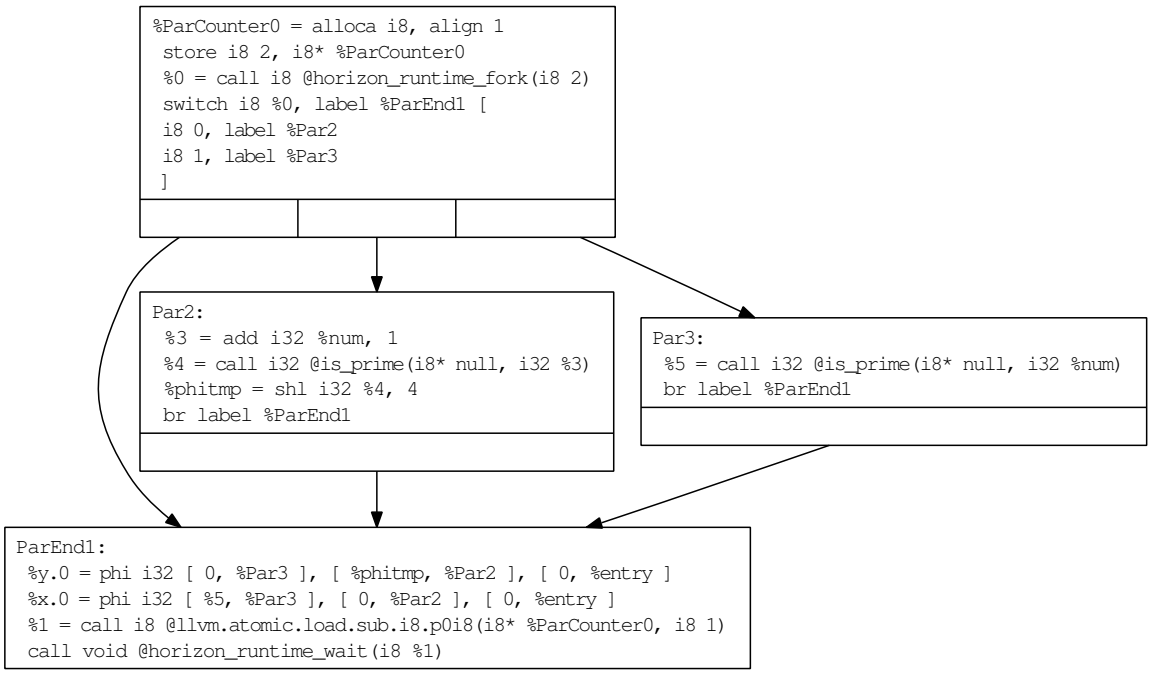


Figure 6.2.: Sample program with -O1. Note the phi transformation.

```
⟦ par {                    ⟧ ↦ ⟦ %ParCounter0 = alloca i8
     <statements0>                 store i8 2, i8* %ParCounter0
  } {                              %1 = call i8 @horizon_runtime_fork(i8 2)
     <statements1>                 switch i8 %1, label %ParEnd1 [
  }                                i8 0, label %Par2
                                   i8 1, label %Par3
                                   ]
                                Par2:
                                   <statements0>
                                   br label %ParEnd1
                                Par3:
                                   <statements1>
                                   br label %ParEnd1
                                ParEnd1:
                                   %2 = call i8 @llvm.atomic.load.sub(i8* %ParCounter0, i8 1)
                                   call void @horizon_runtime_wait(i8 %2) ⟧
```

Mapping 6.1: Parallelism first attempt

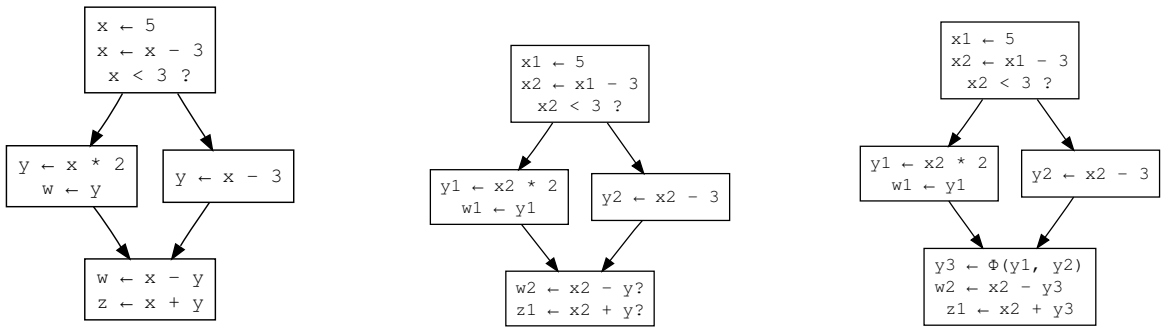
Phi (Φ) functions are a part of single static assignment (SSA) form. In SSA form each register can only be assigned once. Variables assigned multiple times are split into multiple variables with subscripts; for example in the following listing:

```
y := 1
y := 2
x := y
```

The initial assignment on line 1 is masked by that on line 2 and never gets used. SSA form would rewrite this as:

```
y1 := 1
y2 := 2
x  := y2
```

Written like this it is trivial for the compiler to discover that the assignment on line 1 (`y1`) is never used. As a more complex case consider the control flow graph in figure 6.3(a). This is a simple routine that performs some arithmetic on four variables and contains an if/else branch. Converting this to SSA form yields something similar to figure 6.3(b). Notice the problem in the last block - `y` here could be `y1` from the left block or `y2` from the right. SSA form provides a construct for encoding this information, called a phi (Φ) function. This is a grouping function - the value of a phi function could be any of its parameters depending on how the current block was reached (the predecessor block). Its use is demonstrated in figure 6.3(c).



(a) Initial sample CFG (b) CFG in SSA form (c) CFG with phi function

Figure 6.3.: SSA & phi transformation example, courtesy of [16]

LLVM, being based wholly on SSA form, has a special **`phi`** instruction. Its use can be seen in figure 6.2, which is the same sample program from listing 6.1 but compiled with a small amount of optimisation. LLVM has added two phi instructions to the last block:

```
%y.0 = phi i32 [ 0, %Par3 ], [ %phitmp, %Par2 ], [ 0, %entry ]
%x.0 = phi i32 [ %5, %Par3 ], [ 0, %Par2 ], [ 0, %entry ]
```

The first can be translated into English as "%y holds the value of 0 if control came from block %Par3, %phitmp if control came from %Par2 or 0 if control came from %entry".

When compiling to native instructions the phi functions are not actually implemented - instead they are markers for the compiler to allocate all variables in the phi function to the same register or memory slot. This causes my implementation problems, as control will come from *all* predecessor blocks, not just one. The compiler produces code that will break when both threads are run in parallel - shown in figure 6.4. The important instructions are the last two lines in both the left and right blocks. Both `%eax` holding `%y` and `%esi` holding `%x` are live out of *both* blocks. The lower block, ".LBB1_3", assumes this and simply adds them together to produce the result (this is because of a code-hoisting optimisation that pushed the calculation of both variables to the middle two blocks).

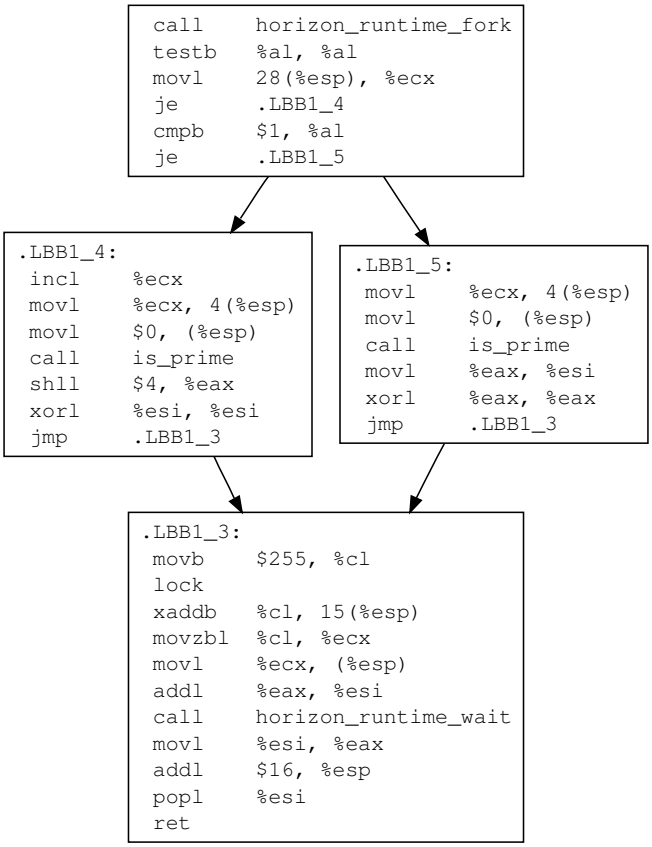


Figure 6.4.: Sample program fully compiled with -O1.

Obviously, if these branches are run in parallel we would have to selectively take register values from each branch (`%eax` from the left branch and `%esi` from the right) at wait/sieve time. This is impossible in the general case.

I attempted to remedy the situation by adding volatile `nop` instructions at the end of each block and telling the compiler that they clobbered all registers to try and force it to spill temporary results back to the stack before the branches join. This was unsuccessful. The important lesson learned was never to break any assumption the compiler makes.

6.2. Final model

Because arbitrary Horizon code is allowed in `par` blocks they must be implemented in such a way that accessing local variables is possible. One method considered was to create a function for each subblock inside a `par` block with a parameter for each local variable used. This was rejected as it would complicate the compiler and the outputted code, would be inefficient and would cause problems with updating any changed local variables.

The final solution was to remove the ability to insert arbitrary Horizon code in `par` blocks. Par blocks are now collections of `call` instructions, exemplified below, the rewritten version of the previous sample program (listing 6.1).

```
call %a *%num.add 1
par {
    call %y is_prime %a
    call %x is_prime %num
}
call %r *%y.shl 4
call %r *%r.add %x
ret %r
```

Mapping 6.2 shows the new mapping to LLVM bitcode. Each function except the n'th is dispatched via a call to the new runtime function `split()`. This function takes the same counter variable as before and increments it when it spawns a new microthread. It takes a function address f and a pointer to a variable that will store the return value of the function. It takes a variable number of parameters, which are passed to f.

The n'th `call` is done in the current thread as a standard function call. After this the second runtime function, `join()` is called. This waits until the counter variable given reaches zero (when all created threads have completed) at which point it returns.

$$\left[\!\!\left[\begin{array}{l}\texttt{par \{}\\ \quad\texttt{call <ret0> f <params0>}\\ \quad\texttt{call <ret1> g <params1>}\\ \texttt{\}}\end{array}\right]\!\!\right] \mapsto \left[\!\!\left[\begin{array}{l}\texttt{\%ParCounter0 = alloca i8}\\ \texttt{store i8 2, i8* \%ParCounter0}\\ \texttt{call void @horizon\_runtime\_split(i8* \%ParCounter0,}\\ \quad\texttt{@f, i8* \%ret0, i8 <\#params0>, <params0>)}\\ \texttt{\%ret1 = call i32 @g(<params1>)}\\ \texttt{call void @horizon\_runtime\_join(i8* \%ParCounter0)}\end{array}\right]\!\!\right]$$

Mapping 6.2: Parallelism final model

This provides a much cleaner and smaller mapping than 6.1 and has another advantage - the same thread that started executing the `par` block will always be executing at the end. In the previous implementation it was a race condition as to which thread reached the `wait` function last and would end up being the main thread out of the `par` block. Here it is defined and deterministic, and means the entire thread stack does not have to be copied on every microthread creation.

For reference, the code for the microthread runtime (`horizon_runtime_split` and `horizon_runtime_join`) is available in the unmarked appendix B.

6.2.1. Parametric qualifiers

It was noticed that a construct similar to a "parallel `for`" would be difficult to achieve with the current model. A parallel `for` is a `for` loop with each iteration run in parallel. With the current model this would be impossible.

A new construct was therefore added to the language to make this possible - parametric qualifiers. A parametric qualifier precedes any `par` block, causing it to be repeated a set number of times; it has the following syntax:

```
par [%r = 0..%end step 1] {
  ...
}
```

The entire block is executed in parallel for each value of `%r` in the range `0` to `%end`, with `%r` being incremented by 1 each time. The register `%r` is live just for the scope of the `par` block. This provides a simple and clean method of spawning a specific number of microthreads.

7. Implementation

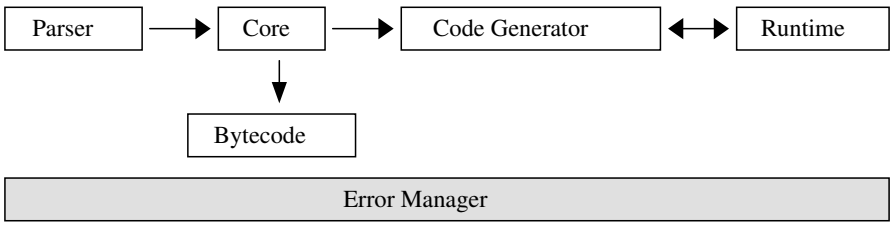


Figure 7.1.: Block architecture of Horizon

The Horizon codebase is composed of five distinct modules; The parser, the "core", the code generator, runtime and bytecode serialiser. These all have access to the Error Manager, which provides properly formatted error messages and warnings.

7.1. Parser

With Horizon I had the choice to either use a prefabricated bottom-up token based parser or a custom top-down recursive descent parser. Bottom-up parsers such as GNU Bison (YACC) are excellent for creating parse trees from simple grammars. However I wanted to create the abstract syntax tree directly in the parser, bypassing the parse tree stage, which requires lots of extra code that can get confusing in a YACC file. I also agree with Clang's view that custom recursive descent parsers are faster and more maintainable [3].

A large percentage of the parser is code to produce readable and understandable error messages; for example instead of a generic message such as

```
EOF while searching for token FAT_COMMA
```

The parser would output

```
In binding list (<...>), expected '=>' after binding target `a', but read `,'.
```

This graceful error handling is tedious and difficult in YACC.

I therefore created a custom recursive descent parser for Horizon using GNU flex (the GNU version of the UNIX utility `lex`) to create the scanner/tokeniser. The simplicity of Horizon made this easy; not many disambiguation rules were required. The entire parser, producing an abstract syntax tree is implemented in one file, `Parser.cc`, in 1864 lines of code - including copious error management code.

7.1.1. Identification and resolution

Once the AST is created internal links need to be established between symbol declarations and their references. In languages such as C this is a one-pass task, using a simple stack tracking the declarations available in the current scope. When a reference to a declaration (such as a function call) is encountered the stack is searched for the given identifier. If it is not found, an error is generated. This means that all identifiers must be declared before they are referenced (hence "forward declarations" in C and C++).

Listing 7.1: AST representation of the mov instruction (whitespace & comments truncated)

```
class AstMovInsn : public AstInsn {
public:
        AstMovInsn() : m_operands() {}
        ~AstMovInsn() {}

        virtual void Print(std::ostream &stream);
        virtual std::string GetName() {return "";}

        std::list<AstOperand*> &GetOperands() {return m_operands;}

        virtual void Identify(SymbolTable &symtab);
        virtual void Resolve(SymbolTable &symtab);
        virtual void ToCore(horizon::Module &module,
                            horizon::FunctionSignature *fn,
                            horizon::Block *block);

private:
        std::list<AstOperand*>   m_operands;
};
```

Listing 7.2: Core representation of the mov instruction (whitespace & comments truncated)

```
class MovInsn : public Instruction {
public:
        MovInsn(Register *dest, Value *src);
        virtual void Print(std::ostream &s) const;

        const Register *GetDest() const {return m_dest;}
        const Value *GetSource() const {return m_src;}
private:
        Register *m_dest;
        Value *m_src;
};
```

Like Java, Horizon is designed so that a function can be defined later in the input file than where it is first referenced - this requires two passes (called *identification* and *resolution*) and a special stack structure.

The class `SymbolTable` keeps track of all symbols. It is a "stack with a memory", for want of a better phrase. During the identify pass the stack is populated as normal with symbol names and their associated declaration objects. Scopes are pushed and popped from the stack so that symbols are only available according to standard scoping rules. The difference is that when a scope is popped, the `SymbolTable` does not delete it as a normal stack would. Instead it maintains a pointer along with a unique name that was given when the scope was pushed.

When the Resolve pass starts, the symbol table is completely full. Scopes are "retrieved" from the stack by supplying it with the same identifier as was used to push the scope in the Identify pass. This ensures that all symbols defined at any point in the file are known at resolve time, and that proper scoping rules are still followed.

7.2. Core

The Core module is essentially just a representation of the program along with ancillary functions such as dumping a textual representation to an output stream. The Core representation is a more usable form of the abstract syntax tree; this can be seen in the example listings 7.1 and 7.2.

The AST version of the `mov` instruction contains a list of operands; the Core version contains the much more specific "dest" and "source" operands only, and also stipulates that "dest" must be a register (but that "source" can be any value).

The AST gets converted by another pass (`AstCore.cc`) and generates Core datastructures using a similar interface to LLVM's IRBuilder. The respective class diagrams are appendicised in sections A.2 and A.1.

Types are handled specially in Core. Each derived type has an associated `TypeConstructor` subclass (for

example `ClassTypeCtor` and `VariantTypeCtor`). `Types` are instantiations of `TypeConstructors` with an optional concretion of polymorphic types. For example, in the following code snippet the class `C` has a member variable of polymorphic type `'a`.

```
class C {
  variable v : 'a
}
```

In this case the type constructor would have `v`'s type as the generic placeholder `'a`. A `Type` instantiation would contain a reference to the `TypeConstructor` and a mapping of the `'a` polymorphic type to a concrete type such as `i32`. The concrete type would be defined either at parse time in the case of a binding list - `alloc %a : C<'a => i32>` or at code generation time in the type inference pass - see section 7.4.1.

7.3. Bytecode

An important feature of a bytecoded language is the ability to produce bytecode. In Horizon's case this is just a serialised form of the Core datastructures. The format had several aims:

- Must be very fast to read; Write speed can be slower than read speed.
- Can be very small; Compact representation.
- Optimise the common case the most; The common case is reading the entire tree in-order. Random access is not expected.

The bytecode format is binary as opposed to text. Text is verbose, large and unneeded - Horizon already has a textual representation. The Core data structures make up a directed graph. It may have cycles, which means that the order in which nodes are written is important - as is the method of referring to other nodes so no duplication takes place yet the exact graph is replicated.

Therefore the graph is written to disk in a depth-first manner, starting with the `Module` instance of which there is guaranteed to be only one. When a node is written to the stream it is given an implicit numeric ID (n where it is the nth item written to the stream). A mapping between a pointer to the node and its ID is then stored by the bytecoder. When an attempt is made to write that same node to the stream again, it is substituted by a reference to ID n.

The tree is serialised as a sequence of *opcodes* and *integers*. Opcodes indicate the beginning of a node and are 8-bit integers with bits 0..6 describing the node type and bit 7 as a "reference bit". If the reference bit is set, the node has already been written to the stream and immediately following the opcode will be an integer specifying its ID.

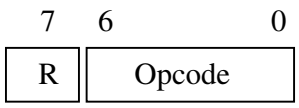


Figure 7.2.: Opcode format

Each node has its own format; there is a function for reading and writing every kind of node. For example, listing 7.3 shows the "Write" routine for the `bc` instruction. Ignoring the debugging macros the `WRITE_PREAMBLE` macro checks to see if the instruction `i` has been written to the stream already. In this case it writes a reference and returns. Else it writes an opcode. The condition is a `Register`, the TrueTarget and FalseTargets are `LabelInsns` - Write() has an overload for each of these. An integer is used to indicate the presence of the "FalseTarget" member - if it is 1, no false target exists.

7.3.1. ULEB128

The bitwidth of integers written to the stream is important. Most integers written are small (< 64), however some are larger with no arbitrary limit except the native integer widths of 2^{32} or 2^{64}. Writing a 64-bit integer

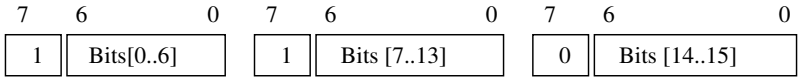
Listing 7.3: Bytecode write function for the `bc` instruction

```
bool Bytecoder::Write(const BcInsn *i) {
        DEBUG_START(BcInsn);
        WRITE_PREAMBLE(BcInsn, i);

        bool b = Write(i->GetCondition());
        if(b) b = Write(i->GetTrueTarget());
        if(b) {
                if(i->GetFalseTarget()) {
                        WriteInt(0);
                        b = Write(i->GetFalseTarget());
                } else {
                        WriteInt(1);
                }
        }

        DEBUG_END(BcInsn);
        return b;
}
```

where an 8-bit would do is very space wasteful. A format called ULEB128 - Unsigned Little-Endian Base-128 - was designed for the DWARF debugging file format [6]. It provides a variable-sized integer by using one bit of every byte as a signal to either carry on reading or to stop. For example to encode a 16-bit integer:



16 bits requires 3 bytes to store. However this is in the worst case - if the most significant bit is set. Bytes are added while the integer to be written is nonzero. When all remaining bits are zero the encoding stops, allowing a very compact (1-byte) representation for values < 128, a 2-byte representation of values $< 128^2$, and so on.

7.4. Codegen

The Codegen module takes Core datastructures and generates LLVM bitcode. It operates on whole functions at a time; one begins by creating a `FunctionInstantiation` object from a `FunctionSignature` object. Figure 7.3 shows the hierarchy of classes related to functions.

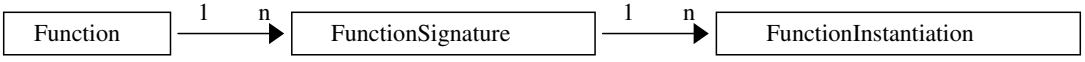


Figure 7.3.: Class diagram for functions - each is a "has-a" relationship

In Horizon, functions can be polymorphic. That is, they can share the same name but have different return or parameter types - referred to as the *function signature*. The `Function` class is a wrapper for multiple `FunctionSignatures` that share the same name.

A `FunctionSignature` has a set of registers and an associated `Block` of code. It can however contain polymorphic types. Depending on how the function is called, these types may change. So, for every `FunctionSignature` there are many `FunctionInstantiations` that have a full mapping of polymorphic types to concrete types. `FunctionInstantiations` are what the code generator works upon.

7.4.1. Passes

Type inference

The first pass takes care of inferring types for each register. Horizon is strongly typed, so each register must have a type and this cannot be changed throughout its lifetime. It is through this that type safety is obtained. When a function is initially instantiated polymorphic types may still appear - these must be substituted for their concrete type. Functions may return polymorphic values - these must be inferred also. The inference rules are described below, with $T(x)$ indicating the type of x.

Note that the = below are enforced equivalences, not assignments - flow of concretions is two-way.

`alloc %r : ty`	$\mapsto$	$T(r) =$ `ty`
`bc %c, l1, l2`	$\mapsto$	$T(c) =$ `i1`
`call`	$\mapsto$	See explanation below
`sext %d : ty, %s`	$\mapsto$	$T(d) =$ `ty` (zext, icast, ucast, dcast identical)
`load %r, var`	$\mapsto$	$T(r) = T(var)$ (`store var, %r` identical)
`load %r, %x.mem`	$\mapsto$	$T(r) = T(x.\text{mem})$ (`store %x.mem, %r` identical)
`mov %d, %s`	$\mapsto$	$T(d) = T(s)$
`ret %x, %y`	$\mapsto$	Any polymorphic return types inferred as $T(x), T(y)$

When a `call` instruction is seen, the most relevant `FunctionSignature` for the callee is picked and a `FunctionInstantiation` created for it if it does not already exist, and the type inference pass run on it. This causes all polymorphic return types to be concreted and allows the original function to infer its register types correctly in one pass.

Because `FunctionInstantiations` are created for every function call encountered during the type inference pass, it is only necessary to manually create one instantiation - for the entry or main function. This also provides implicit dead code elimination for functions that are never used.

Escape analysis

Escape analysis is the inspection of a function f to discover which of its registers' lifespans are wholly contained by f and which "escape" and cannot be destroyed at the end of f. The interesting registers in particular are those that are created by an `alloc` instruction. It is far quicker to allocate a register on the stack if at all possible, rather than with the garbage collector. This can only be done however if it is known that the allocated register has ended its active life at the end of it's `alloc`cing function.

It should be noted that the only registers that can escape are those containing reference types. Those containing basic types such as integers or floats are passed by value so the notion of "escaping" does not apply.

These registers must therefore be traced through the function, through all possible branches and through all other functions that may be called before the containing function ends. Given that it is impossible to know at compile time which branches will be taken and which not, it is safer to ignore all branches and assume that every instruction in the function can be visited at least once. This will only create false positives, not false negatives, so is safe.

Therefore we can iterate through the function block keeping track of which registers' values influence which others, in order to discover the data flow graph of `alloc`ed registers.

This is inefficient however, because as soon as a function call is encountered it must be assumed that all registers passed as parameters will escape. Therefore the very simple instruction "`call *%a.add %b`" on integers (or an integer-like user defined class) would class both `%a` and `%b` as escaping[1], even though with such a function they would not.

It might be better therefore to trace a function's parameters too, and to have a function on a `Function Instantiation` object to query the "escape state" of each parameter. This gets even more complex however, as an "escape state" may not just be a simple boolean true or false. Consider the following code snippet:

```
class C {
        variable v : C
        variable v2 : C
}

function () f(%x : C) {
```

[1] Here `%a` is passed as the `%this` parameter.

```
        load %tmp, %x.v
        call function_that_escapes %tmp
        ret
}
```

Here the "escape state" of the parameter `%x` is multi-faceted. Only one member of `%x` (`v`) escapes. The other does not and is safe to allocate on the stack. It could be argued from both sides that `%x` does or does not escape. The answer is to create an `EscapeState` class that can denote the simple case of an entire register escaping and the more complex case of subcomponents of that register escaping.

Below is the rule set for escape analysis of specific instructions in a predicate-logic style. Any instruction that does not appear has no meaning in the escape analysis pass. The ruleset is subject to the following assumptions:

- The function $E(n)$ denotes the escape state for register n.
- The function $P(f, n)$ gets the escape state for parameter number n of function f.
- The function $M(m, s)$ gets the escape state for member variable m of state s.

`alloc %r : ty`	$\mapsto$	$E(r) = \varnothing$
`call () f (%p1,...,%pn)`	$\mapsto$	$\forall i \in 1..n \mid E(\mathrm{p}i) = P(f, i)$
`sext %d : ty, %s`	$\mapsto$	$E(d) = E(s)$ (zext, icast, ucast, dcast identical)
`load %d, %s.mem`	$\mapsto$	$E(d) = M(mem, E(s))$
`mov %d, %s`	$\mapsto$	$E(d) = E(s)$
`ret %r`	$\mapsto$	$E(r) =$ **true**
`store %d.mem, %s`	$\mapsto$	$M(mem, E(d)) = E(s)$
`store global, %s`	$\mapsto$	$E(s) =$ **true**

With this algorithm the variables stored on the stack can be maximised and the variables stored on the garbage-collected heap minimised.

Code generation

The final pass actually generates the LLVM bitcode. This process is detailed in the following subsections.

7.4.2. Constructs

References to derived types are implemented as LLVM pointers. The pointers themselves are immutable.

Classes are generated simply as LLVM `structs`. Every class member is added as a struct member; in the case of virtual functions (inherited and overridden functions) a vtable pointer is added at the start of the struct. This is currently unimplemented.

Interfaces do not have an associated LLVM type and at the time of project handin were unimplemented.

Variants use an LLVM union wrapped in an LLVM struct to provide a type tag. The type tag is checked dynamically at runtime. Variants are currently unimplemented.

7.4.3. Instructions

alloc Assuming escape analysis is complete, the type to allocate is checked against the basic types. If the type is basic, the instruction is the equivalent of "**mov** `dest, 0`". Else space is reserved either on the stack or the garbage collected heap if the object escapes.

Classes have all their members initialised recursively - at the end of an alloc instruction all the object's member variables will be valid - no references will be **null**.

b Translates simply to a llvm **br** `target`.

bc Translates to an LLVM conditional branch: **br** `%c, truetarget, falsetarget`.

call In the case of a standard call or an object member call, this translates to an LLVM **call** instruction. For codegen simplicity all functions, member functions or not, take a "this" parameter as their first argument. Therefore both of these type of calls are identical.

Indirect calls, that call a register value (of type `System.Runnable`) extracts the special values "runnable.this" and "runnable.function" from a **`interface`**`` instance, and constructs an LLVM \lstinlinecall! from them.

Casts The **`trunc`**, **`sext`** and **`zext`** instructions translate into their direct LLVM counterparts, with some static type checking. The upcast, downcast and interface-cast instructions are currently not implemented.

load The offsetted load for arrays extracts the "array.ptr" member from the source array struct, indexes into it to find a pointer to the member required, dereferences it and loads that value into the destination register.

The field-access version of **`load`** finds the index of the required member in the class struct and uses LLVM's `getelementptr` instruction to obtain a pointer to it. The pointer is dereferenced, and the value stored into the target register.

mov The easiest instruction - the source register is stored into the target register with a single LLVM **`store`** instruction.

ret These are translated simply into LLVM `ret` instructions.

store Operates in the opposite way to **`load`**.

7.5. Runtime

At the time of writing the runtime is very small, just 5 functions. It will grow as the language matures. Below is a table of the current runtime functions.

`horizon_runtime_split`	Causes a microthread to be spawned. Documented in section 6.2.
`horizon_runtime_join`	Causes the thread to wait until all microthreads in an earlier `split` have finished; See 6.2.
`horizon_runtime_new_stack`	Called by `split` when a new stack is needed for a spawned microthread.
`horizon_runtime_print_int`	Prints an integer to the standard output.
`horizon_runtime_print_array`	Prints an array to the standard output.

8. Evaluation

Evaluation of the deliverable with respect to the specification given in section 3.1 requires both a functional evaluation and performance comparison.

8.1. Functional

Points 1 and 5 of the specification (Language safety and paradigm support) are answerable solely by the design of Horizon. The language does *not* allow casting from numbers to reference types or vice versa, does not allow inline assembler or any other unsafe tool. Because of this, the language is safe. It does allow NULL references and thus dereferencing a NULL pointer, but this is catchable and is allowed in the specification.

Horizon supports all required paradigms - procedural, imperative and object-oriented. Indeed, procedural and imperative are subsets of the object-oriented paradigm, which is pervasive throughout Horizon. Everything is an object and can have member functions - (like Ruby), even integers. There is not the C#, C++ or Java-like distinction between basic types and derived types in that basic types cannot have member functions.

Although not necessarily related to the specification, the following table details the parts of Horizon that are complete at the time of project hand in and the parts that are incomplete:

	Parse	**Core**	**Bytecode**	**Codegen**
Classes	☑	☑	☑	☑
Interfaces	☑	☑	☑	☑
Variants	☑	☑	☑	☐
Integer constants	☑	☑	☑	☑
Floating point constants	☑	☑	☑	☐
String constants	☑	☑	☑	☑
Par blocks	☑	☑	☑	☑
Parametric qualifiers	☑	☑	☐	☐
Alloc	☑	☑	☑	☑
B	☑	☑	☑	☑
Bc	☑	☑	☑	☑
Call	☑	☑	☑	☑
Casts	☑	☑	☑	☐
Labels	☑	☑	☑	☑
Load	☑	☑	☑	Partial
Mov	☑	☑	☑	☑
Ret	☑	☑	☑	Partial
Store	☑	☑	☑	Partial
Garbage collection	-	-	-	☐
Thread context switching	-	-	-	Partial (32bit only)
Virtual methods	-	-	-	☐

What was implemented was enough to run the Dhrystone test benchmark (see next section) - the most complex program required. This involved some of the more advanced aspects of the code generator such as class layout generation and escape analysis. These were both implemented fully.

The thread context switching code was written for 32 bit x86 systems as the testbed bare bones OS I was using ran only in 32 bit mode.

8.2. Performance

8.2.1. Quicksort

Divide-and-conquer algorithms are an obvious choice for testing multithreading due to their easily segmentable nature. Of the many divide and conquer algorithms, quicksort is one of the most used and the most simple. The test algorithm uses the in-place version of the quicksort algorithm to avoid creating many copies of the source array. Random test arrays of differing sizes were created and their sort time measured.

Quicksort's control flow graph can be visualised in the average case as a binary tree. If we spawned a new microthread at each node in the tree (on each execution of the main quicksort function) we would end up with 2^n threads. This can get large very quickly, especially as the number of items in the array must be large to obtain a measurable time. I therefore implemented a cutoff depth - after n recursions no more microthreads will be created. For this test, n has a maximum of 10, so up to 1024 threads can be created. n can be varied to see the extent that creating more microthreads has on the execution time.

Results

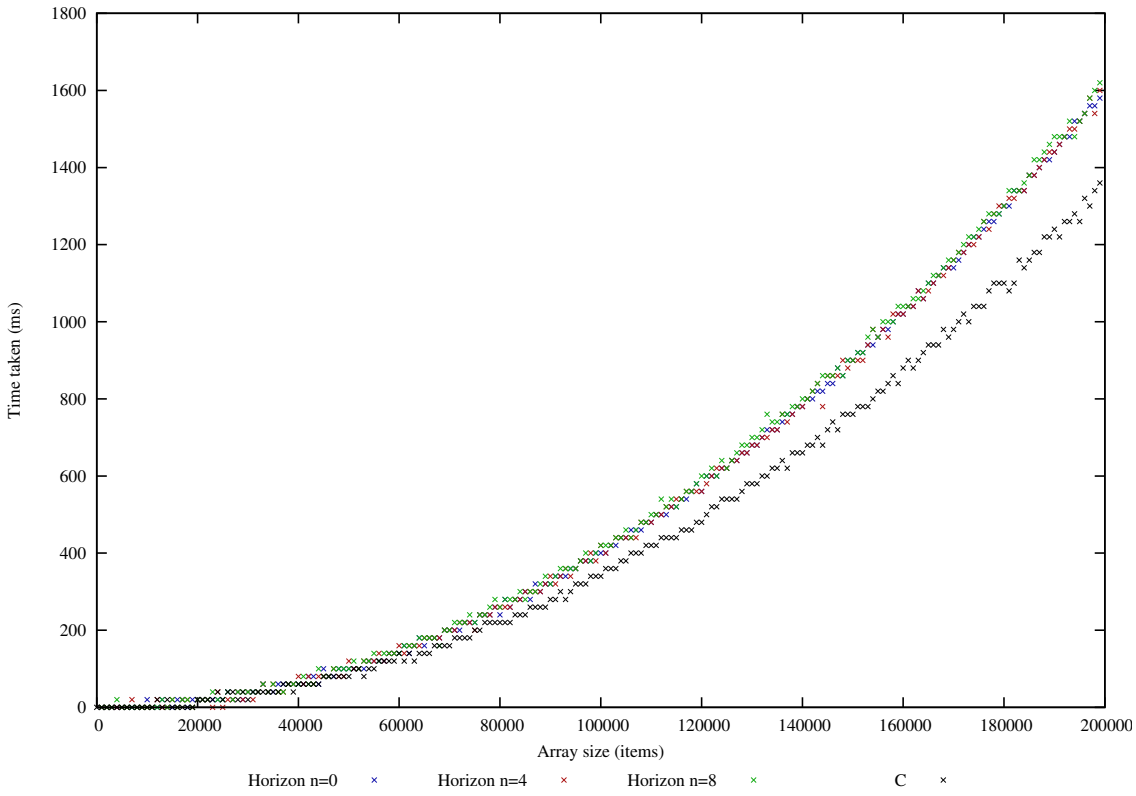


Figure 8.1.: Quicksort performance for Horizon with 2^n µthreads and single threaded C.

The results are shown in figure 8.1. In this test one can see that changing the number of microthreads used has *no visible effect on the result*. Values of n shown are $\{0, 4, 8\}$, which equate to $\{1, 16, 256\}$ microthreads. 1024 microthreads were also tested, however the result was exactly the same as for other values of n and was omitted from the graph for fear of cluttering it.

Horizon has a larger leading constant than the C code, but it is comparable in speed.

Criticism

Quicksort is not an indicative test of a normal workload. It is highly compute intensive and involves massive amounts of recursion. A more relevant test might be a standard CPU benchmark.

8.2.2. Dhrystone

Dhrystone is a CPU benchmark and was introduced in section 5.3.2 for measuring context switch performance. It is a synthetic benchmark originally developed in 1984 and was designed to statistically mimic a common set of programs. It contains variable assignments, integer arithmetic, string manipulation and array manipulation.

The name is a play on the name of another popular benchmark "Whetstone", which also measures floating point performance.

Scientific method

The version of Horizon at the time of project completion is not fully able to manipulate strings, and neither 2D arrays. Therefore these parts of the benchmark were stripped, leaving variable assignments and integer arithmetic.

The benchmark algorithm was transcribed into Horizon and a bare-bones harness set up to run and time the program - this is based on the harness used in section 5 [9] so no overheads were involved.

Both the C and Horizon versions of the benchmark are appendicised - listings C.1 and C.2 respectively. Note that what was tested was a rewrite of the original code, maintaining the exact algorithm - the original code had extraneous content and was untidy.

The code was set up to run a fixed number of iterations of the main loop (5000000). To test microthread performance, the Horizon code was modified to spawn variable numbers of microthreads - each ran their own copy of the algorithm but ran $\frac{5000000}{n}$ iterations in order to keep the total iteration count constant and ensure comparability.

Tests were run in a randomised x-order to remove systematic errors, and 15 repeats were taken for each data point.

Results

Tests were run for numbers of microthreads in the range [0..1024). This was compared against the equivalent C performance. Because the C code is not threaded at all, the graphs show the initial 15 repeats at x=1 and the mean average of these extended to all $x > 1$, joined by a line to aid visual comparison with the Horizon results.

The expected result if Horizon were as fast as the C equivalent would be a curve starting at the same point as the C performance curving upward slightly as the (small) overhead of more threads increases run time. A lower value is better for this graph - a successful outcome is a performance of better than or equal to the C equivalent.

The full result set is shown in figure 8.2.

As the graph is rather messy and no data trend is visible in the higher part of the domain, figure 8.3 truncates the x-axis to the range [0..100].

Referring to figure 8.3, the y-axis measures the time taken to perform a fixed number of iterations in milliseconds; a lower value is better than a higher value. Two important results can be seen:

- With the exception of two outliers the Horizon times are significantly lower than that of the C code, showing that for this benchmark Horizon performs better than C.
- Although the results are somewhat scattered and have little correlation, and without serious statistical analysis, it can be observed that increasing the number of microthreads does not adversely impact the performance of the system.

These results directly corroborate the second claim made in the project hypothesis (section 3.1) - "... such a language does not imply slowness - the speed of such a language can approach that of C.".

It should be noted that these results were obtained by producing LLVM bitcode from both languages; using the Horizon→LLVM compiler and the `llvm-gcc` LLVM C front end. This bitcode was then passed through exactly the same optimisation and compilation passes. For these results the optimisation setting `-O1` was used - this performs multiple passes of the "standard" instruction combining and strength reducing optimisations, but nothing more complex. There is more discussion about this in section 8.2.2, but using the full `-O3` optimisation pass caused the compiler to statically eliminate major parts of the code, causing it to not be representative of the original algorithm.

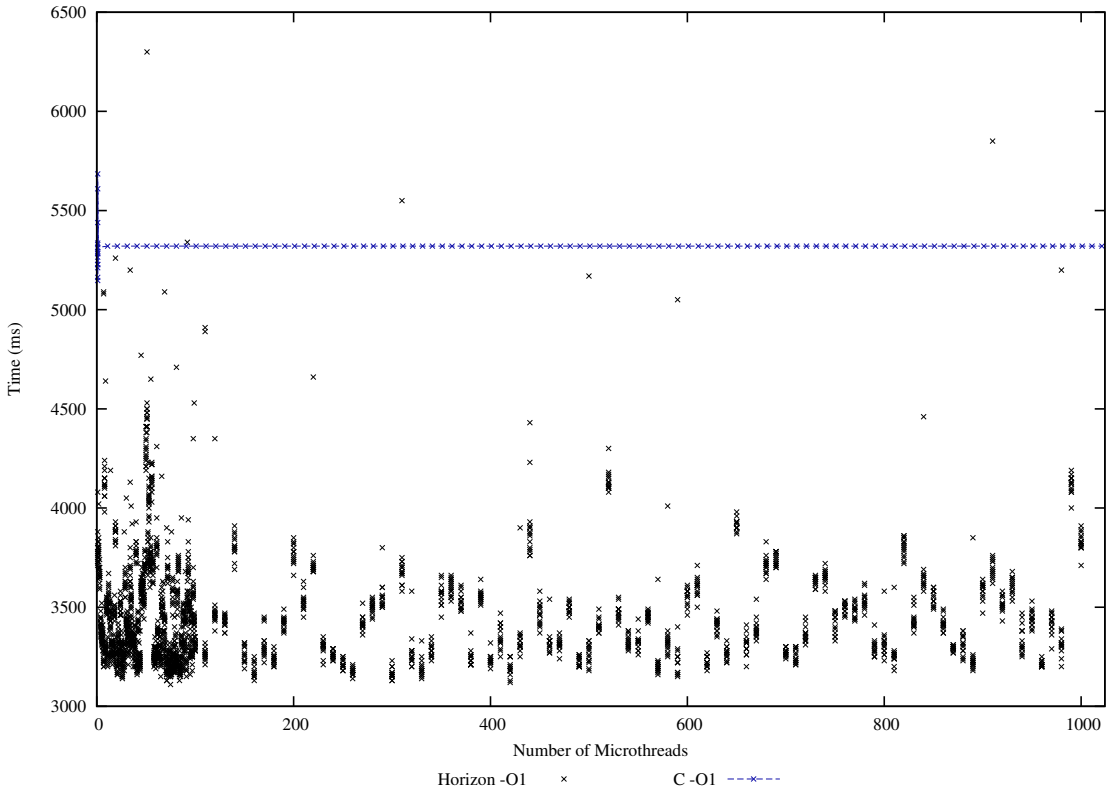


Figure 8.2.: Dhrystone benchmark results for differing numbers of µthreads, against C performance.

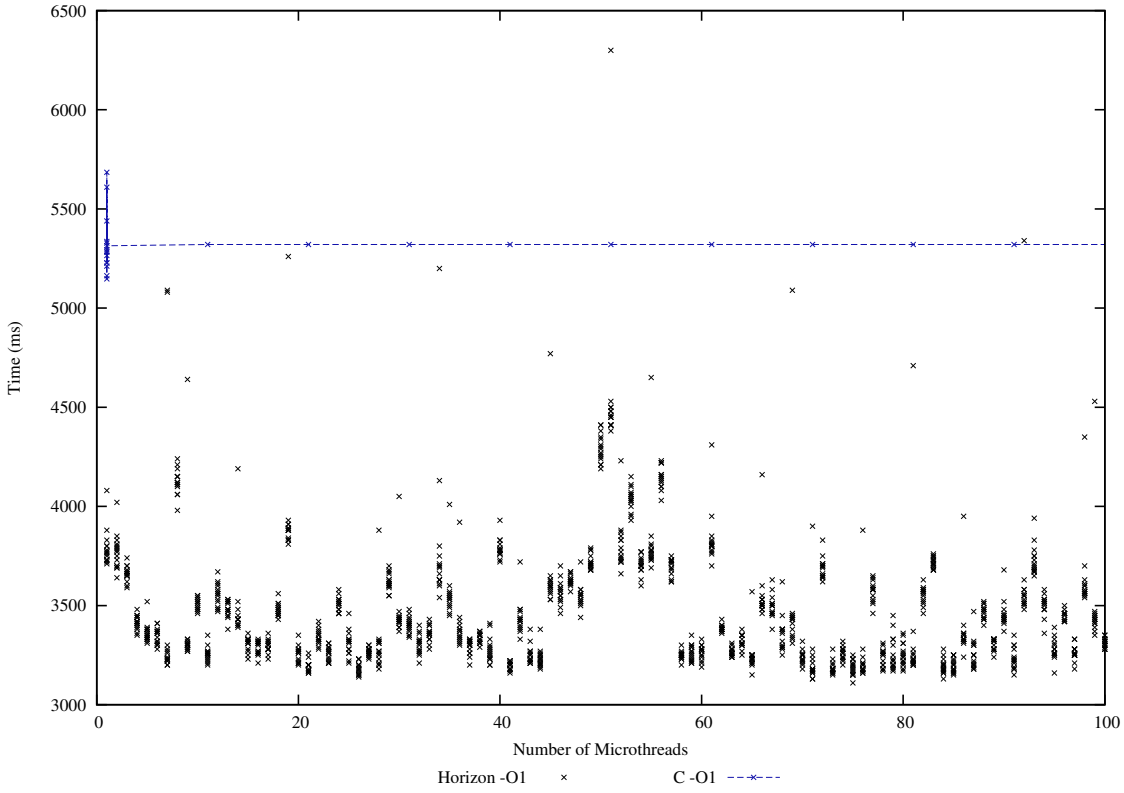
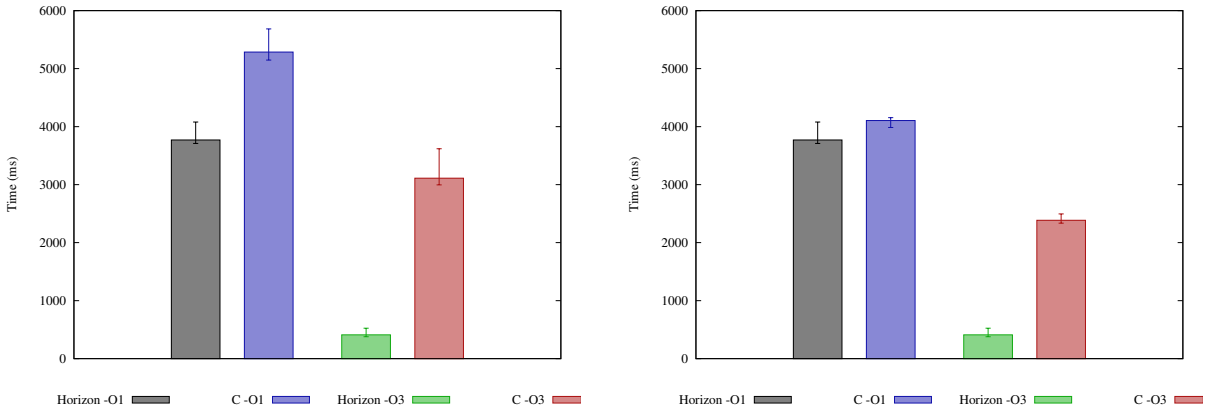


Figure 8.3.: Dhrystone benchmark results, zoomed to [0..100] on the x-axis.

I thought perhaps that the C code might outperform Horizon when the optimisation level was increased; the results of this are shown in figure 8.4(a). The green and red bars were compiled with optimiser arguments "`-O3 -std-link-opts`" which enables LLVM's most powerful optimisations as well as link-time optimisations. The bar height is the median average of 15 repeats, with the error bars marking the maximum and minimum repeat values.



(a) With original Ada-like C code

(b) With procedures → functions.

Figure 8.4.: Dhrystone benchmark results for a single µthread, both with the original C code and a more optimised version.

It can be seen that Horizon became even faster relative to the C code. In Horizon's case the LLVM compiler managed to reduce the algorithm to a single iteration - it noticed that the multiple iterations were unnecessary, hence its impressive speed. In the C case, LLVM seemed not to be able to infer the same - one procedure call alone was not able to be eliminated (a call to "`proc1`").

The C code is written as transcribed from the original Ada version of this benchmark. That is, it uses lots of "procedure"-like constructs instead of the more obvious function form. For example "`f(a, &b)`" instead of "`b = f(a)`". I thought that this use of pointers to mimic Ada constructs might be causing the compiler difficulties with alias analysis. I therefore rewrote the C version to take this into account, thinking it might produce LLVM bitcode more prone to optimisation. The results are shown in figure 8.4(b).

It can be seen that this had some effect, but not near enough to outperform Horizon. LLVM was again unable to eliminate one procedure call and thus the iteration loop (hence the large disparity in time between it and Horizon - Horizon performs 1 iteration, not 5000000!). It seems that for simple integer arithmetic at least, Horizon produces IR that is easier to optimise than the C front end.

Criticism

Dhrystone is an aged benchmark that is susceptible to compiler optimisations. It performs no useful work and does no useful calculation - as such a modern compiler can statically reduce the algorithm to a very small piece of code without any procedure calls or, more importantly, loops - The entire algorithm can be reduced to constant time. This was most noticeable in the Horizon version, but there is no specific reason why the C version could not make the same optimisations.

There are also doubts about its relevance to the operations performed by today's programs as opposed to the programs used to create it in the 1980s.

The benchmark also only tests a small subset of measurable operations - floating point, data and instruction cache miss performance (the algorithm and data is so small it can fit into level 1 cache), inter process communication and I/O are untested by Dhrystone. Granted, this was the main reason for its being selected for evaluation - most of these attributes are not available in the current version of Horizon. It does however mean that the results shown above are not indicative of performance in a "normal usage" or uncontrived situation.

9. Further work

9.1. Multiprocessor

Throughout Horizon's design, multiprocessor systems were thought of. Several factors are important in designing a system that deals with potentially thousands of threads on potentially many (50+) processing cores.

Scalability is obviously important. When dealing with thousands of threads the system must be scalable. This means that scheduling algorithms that are not constant time will start to become costly.

Network topology In current x86 systems every processor can communicate directly with every other, and access all memory controllers with the same cost. As the number of processors increases it will not be viable to increase the number of busses at the same rate - thus a processor network will appear, with some processors only being able to address others transitively through a third processor. This is already the case with Intel's recently aborted "Larrabee" chip and other NUMA designs.

Cache topology Similarly, there are currently three levels of cache with all but the level 3 cache duplicated for each processor (although obviously hyperthreaded processors share their L1 and L2 caches between both hyperthreads). The topology of the caching system is different from the inter-processor communication network and needs to be understood to effectually send threads with similar working sets to processors that share the same cache at some level.

Cache updates X86 has hardware cache coherency. Changing the value in a memory location on one processor immediately affects all others without any user input (c.f. POWER, SPARC). Because of this invisibility to the user the true effect may not be truly realised.

Each cache line is marked if another processor has the line cached too. If a write occurs to a marked cache line, it is immediately flushed to RAM (or more likely L3 cache as it is shared) and a message sent to all other processors to invalidate their cache for that line. In the worst case where the line is no longer in L3 cache, this could require two memory accesses.

Bearing these in mind, future work on Horizon would involve designing and implementing a scheduling and work balancing system. The system would be distributed - each processor running its own copy and not caring about the workloads of other processors. This immediately allows scalability on the number of processors and network topology.

Each processor would have a set of threads that it currently runs. It could schedule them with a very simple limited-priority round robin system, as used in the Linux kernel for years until the completely fair scheduler (CFS) was introduced. Regarding this, Google runs custom versions of Linux with the CFS disabled and the round robin reenabled as CFS (which has $\log_2 n$ time complexity) was hindering their performance with many (1000+) threads running.

Thread migration could be through an attraction-repulsion based algorithm. Threads with similar working sets could be attracted to one another. Threads using a particular memory bank more often than another may be more attracted to processors that give quicker access times to that memory on NUMA systems. Threads could be weakly repelled by one another, so that many threads on one CPU would cause some to migrate to others nearby. Threads could have an "inertia", or attraction to not moving - this could model the performance hit taken by moving to another CPU and starting execution again with a cold cache.

All of these ideas are implementable and are constant time. They take advantage of knowledge about the processor and cache topology and as each processor would use its own data structures, there would be no sharing and so cache updates would be lessened.

9.2. Garbage collection

Garbage collection is required, but was unable to be completed in time for project handin. The ideal garbage collector would be real-time and not "stop-the-world" - such as Limbo's colouring garbage collector.

As a form of "market research" I have asked other hobbyist low level developers what features they would like to see in such a language, and the resounding answer was the ability to manage memory manually - `malloc/free` style. This is completely unsafe in applications, but for a trusted kernel or application could be tolerated. I would therefore add the ability to manually manage memory, and the ability to disallow it.

9.3. High level language

Horizon itself is unwieldy to code in. A high level language that compiles down to Horizon would ease programming, and the raw speed of Horizon could make it a competitor to C in terms of straight line speed. As there are no languages currently targetting Horizon and converting any current language would be difficult due to paradigm differences, there is a "blank canvas" available to design a new language based on modern design principles, programming techniques and paradigms.

9.4. Operating system

One of the real advantages of Horizon, language enforced process isolation, could cause an operating system that is written with Horizon in mind (and enforced that all applications compile down to Horizon) to have extremely fast inter process communication. As most processes on a standard desktop system are I/O bound waiting for user input or transferring data, this could speed up the entire user experience by a large amount.

10. Conclusions

Summary of work done

This project has documented the creation of a new language and compiler suited to systems programming, dispatching with the traditional kernel stack and starting afresh. Unlike other high-level bytecoded languages, this language strives for speed. Many of the safety features of other high level languages such as exception handling have been removed to simplify the generated code. Only enough safety has been retained that a program cannot accidentally or maliciously cause memory that does not belong to itself to be read or written, and cannot manually change the execution location. Because of this, all processes can be stored in the same virtual address space and communicate very efficiently.

The language extends the LLVM compiler framework and as such has a wealth of optimisations and experienced developers at the code generation level. It has been shown to be comparable to the performance of C - better in one benchmark - so is feasible for use in low-level systems development where speed is of the essence. At the same time it supports a wide range of modern programming constructs - object orientation being the most notable. Thus, one can write *fast* code that is also *readable* and *maintainable*.

Horizon as an intermediate language allows the use of any high-level language designed to target it. As in Microsoft .NET where one can use modules written in C# in an F# program, one can do the same with Horizon. This allows interoperability without pinning the user down to one specific language.

Horizon's runtime shows the feasibility of multithreading at the lowest level possible - building the OS on top of a language that is multithreaded instead of building a language on top of an OS that allows multithreading. Pushing threading to the bottommost layer in the OS stack provides several benefits;

- System-wide knowledge of all threads allows efficient load balancing.
- Threads can be tiny - represented solely by a stack pointer.
- Thread switching mechanisms can use supervisor-mode instructions - `cli` and `sti` are efficient ways of providing single-processor locking.

Relating to future work

Horizon has been written with other language designers in mind - it is so simple yet expressive that many current languages could be compiled to it (if they do not rely on pointer manipulation or exception handling) - it does not pin a programmer down to any specific language. While it is simple and assembler-like in appearance, Horizon is also comprehensible because of its relative verbosity. This will help when debugging modules written in different high level languages.

A small experiment towards the end of this project causes Horizon's simplicity as a compiler target to be made apparent. I wrote a very small compiler for a subset of the Scheme[38] language, targetting Horizon. Scheme is a functional, dynamically typed language - quite different to Horizon's statically typed imperative, object oriented paradigm. With just a single day's work, the following simple scheme program compiled to Horizon and then to LLVM;

```
(define (double x)
    (add x x))

(double 4)
```

→

```
function () _main () {
        mov %r2, 4
        call %r3 scm_double %r2
        call *%r3.print
        ret
}

function ('r) scm_double (%x : 'p0) {
        call %r1 *%x.add %x
        ret %r1
}
```

↓

```
define void @_main(i8* nocapture %this) {
entry:
  tail call void @horizon_runtime_print_int(i64 8, i32 32)
  ret void
}

define i32 @_HZN_scm_doubleRC2I2tEEPC2I2tEE(i8* nocapture %this, i32 %x) {
entry:
  %0 = shl i32 %x, 1                              ; <i32> [#uses=1]
  ret i32 %0
}
```

This demonstrates several abilities of Horizon as an intermediate language. In this (granted, trivial) example Horizon's type inference is used to create a statically typed program from one that is dynamically typed. Notice that in the `scm_double` function's definition the return type and parameter type are polymorphic. With less trivial examples this may be less easy, but the proof of concept is still valid.

Secondly it demonstrates the power of building such a compiler atop LLVM. LLVM with optimisations enabled has reduced the main function statically to the value "8", and reduced the `double` function (which is now no longer called) to a simple left shift[1].

Finally it demonstrates how easily targettable Horizon is. In one day's work and only 659 lines of code (of which only 239 pertain to Horizon codegen) a semi-functional compiler is built.

New areas

The ideas behind Horizon could be applied to new areas - for example porting to ARM or POWER, the next two most used chip architectures behind x86. ARM generally creates more low power chips - on these devices resources must be conserved. The feasibility of running a full just-in-time compiler on limited resources is unlikely at the moment, but the field is evolving so fast that soon the embedded market may benefit from the architecture changes that this project proposes.

POWER on the other hand is more often used in high-end servers and specialist, fast computers. Generally these tend to run more custom software and less of a heterogeneous mix of applications, so may not benefit so much from the just-in-time recompilation abilities of Horizon. The IPC speedup may be extremely useful however, as communication throughput is often a bottleneck in server software.

Final conclusion

To conclude, this project has theorised about a change in the way operating systems are designed, has developed a (partial) solution and tested that solution with respect to current implementations. It was found to be faster in one experiment, slightly slower in the other. I conclude that the project was a success,this area is one worth researching more, and large efficiency gains for a large number and range of systems in the future may be the benefit.

[1] For completeness, the second argument to `horizon_runtime_print_int` is the type of the variable to print, which in this case is a 32-bit integer and has the arbitrary type ID 32.

A. Class diagrams

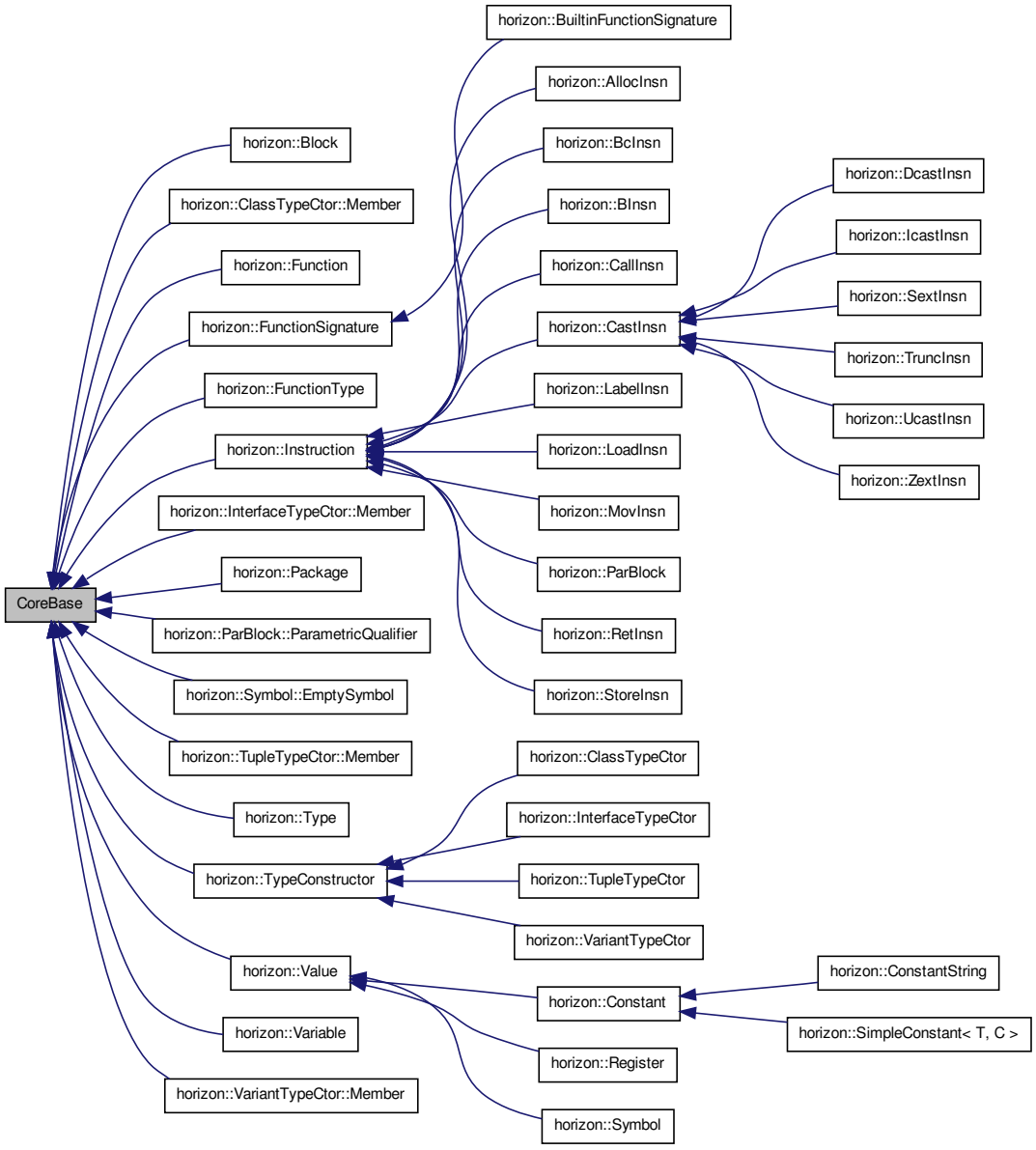


Figure A.1.: Core module class diagram

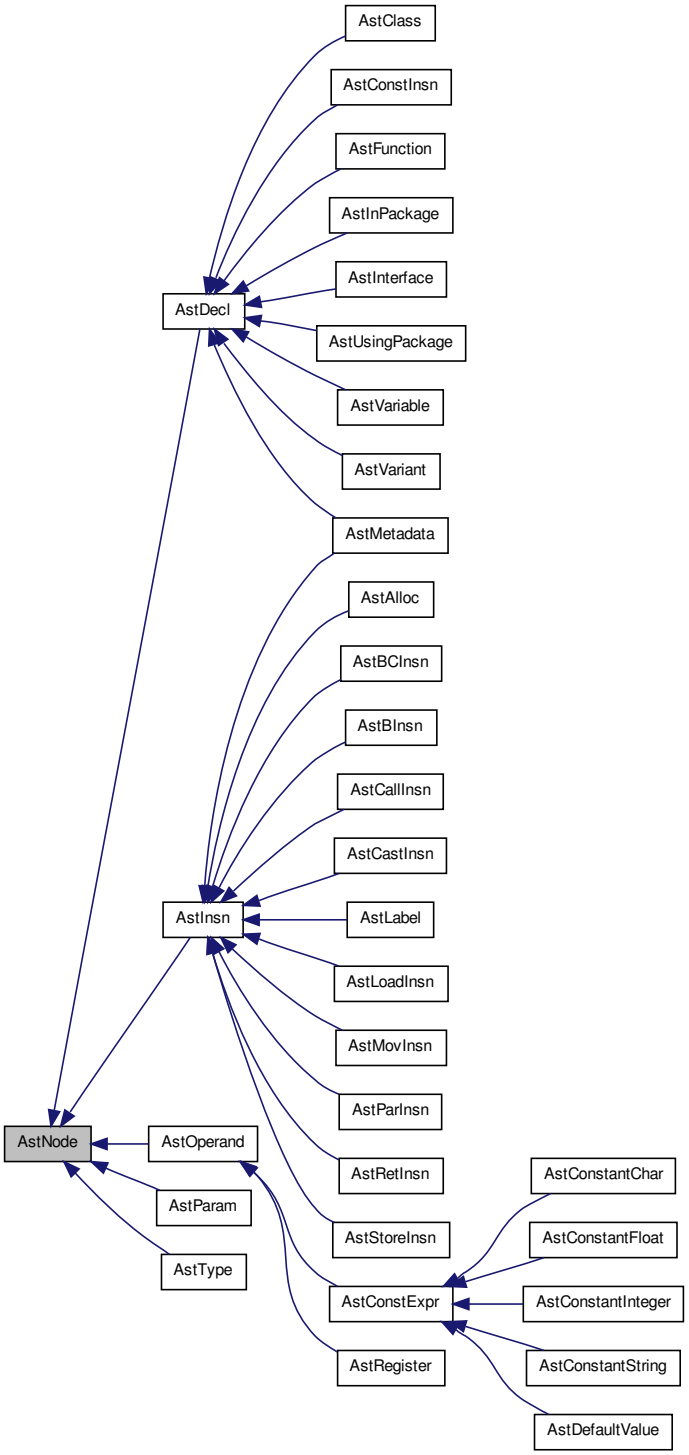


Figure A.2.: Parser module class diagram

B. Microthread runtime listing

```
1  extern thread_buffer_front
   extern thread_buffer_back
3  extern thread_buffer
   extern stack_buffer_front
5  extern stack_buffer_back
   extern stack_buffer
7  extern horizon_runtime_new_stack

9  global horizon_runtime_split
   global horizon_runtime_join
11
   ;;; i8 *ctr : Counter. Should be decremented
13 ;;; when the thread returns.
   ;;; i8 *fn : Function address.
15 ;;; i8 *ret : Put the return value here.
   ;;; i8 nparams : Number of function
17 ;;;              parameters following.
   ;;; ...  : Function parameters
19 horizon_runtime_split:
     ;; Save the address of the start of the
21   ;; params
     lea edx, [esp+4]
23
     ;; Uninterruptible.
25   ;; cli

27   ;; Save current thread state.
     pusha
29
     ;; Change threads and get a new stack.
31   mov eax, [thread_buffer_back]
     mov [thread_buffer + eax*4], esp
33
     inc eax
35   cmp eax, 0x800
     jnz .ok0
37
     xor eax, eax
39 .ok0:
     mov [thread_buffer_back], eax
41
     ;; Get a new stack.
43   mov eax, [stack_buffer_front]
     mov ecx, [stack_buffer_back]
45   cmp eax, ecx
     je .new_stack
47
     ;; A stack was available, so take it.
49   mov eax, [stack_buffer_front]
     mov esp, [stack_buffer + eax*4]
51   ;; Then increment the front location.
     inc eax
53   cmp eax, 0x800
     jnz .ok1
55   xor eax, eax
   .ok1:
57   mov [stack_buffer_front], eax
     jmp .stack_done
59
   .new_stack:
61   ;; No stacks were available, a new one is
     ;; required.
63   call horizon_runtime_new_stack
     mov esp, eax
65
   .stack_done:
67
     ;; Counter reg into non-scratch reg.
69   mov esi, [edx]

71   ;; Function address into scratch reg.
     mov ebx, [edx+4]
73
     ;; Return address into non-scratch reg.
75   mov edi, [edx+8]

77   ;; N-params into scratch reg.
     mov ecx, [edx+12]
79
     ;; Original stack pointer into ebp.
81   mov ebp, esp

83   ;; Copy over parameters.
     add edx, 16
85
     ;; edx = edx + (ecx*4)
87   shl ecx, 2
     add edx, ecx
89   shr ecx, 2
   .loop:
91   test ecx, ecx
     jz .fin
93
     mov eax, [edx-4]
95   push eax

97   ;; Next parameter
     sub edx, 4
99   dec ecx
     jmp .loop
101
   .fin:
103  ;; Interruptible again.
     ;; sti
105
     mov eax, ebx
107  mov ebx, ebp

109  mov ebp, esp

111  ;; Call.
     call eax
113
     ;; Save return value.
115  test edi, edi
     jz .void
117  mov [edi], eax
   .void:
119  ;; Uninterruptible.
     ;; cli
121
     ;; Decrement counter value.
123  lock dec byte [esi]

125  ;; Kill thread and swap to next.
     mov eax, [thread_buffer_front]
127  mov esp, [thread_buffer + eax*4]

129  inc eax
     cmp eax, 0x800
131  jnz .ok2
```

```
133   xor eax, eax
    .ok2:
135   mov [thread_buffer_front], eax

137   mov eax, [stack_buffer_back]
      mov [stack_buffer + eax*4], ebx
139   inc eax
      cmp eax, 0x800
141
      jnz .ok3
143
      xor eax, eax
145 .ok3:
      mov [stack_buffer_back], eax
147
      ;; Interruptible again
149   ;; sti

151   ;; Restore regs.
      popa
153
      ;; Done.
155   ret

157 ;;; i8* counter : Pointer to counter.
    ;;; Wait until *counter gets to zero.
159 horizon_runtime_join:
      mov eax, 0
161   mov ecx, [esp+4]
    .loop:
163   xor al, al
      cmp al, [ecx]
165
      jz .done
167   call horizon_runtime_yield
      jmp .loop
169
      ret
171
    horizon_runtime_yield:
173   pusha

175   mov eax, [thread_buffer_back]
      mov [thread_buffer + eax*4], esp
177
      inc eax
179   cmp eax, 0x800
      jnz .ok0
181
      xor eax, eax
183 .ok0:
      mov [thread_buffer_back], eax
185
      mov eax, [thread_buffer_front]
187   mov esp, [thread_buffer + eax*4]

189   inc eax
      cmp eax, 0x800
191   jnz .ok1

193   xor eax, eax
    .ok1:
195   mov [thread_buffer_front], eax

197   popa
      ret
```

C. Dhrystone listings

C.1. C version

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/times.h>
#include <sys/types.h>
#include <unistd.h>

typedef char String[31];

typedef struct Record_ {
    struct Record_ *ptr;
    int     d;
    int     e;
    int     i;
    //String  str;
} Record;

int     func1(char,char);
int func3(int);
void proc0();
void proc1(Record *);
void proc2(int *);
void proc3(Record **);
void proc4();
void proc5();
void proc6(int, int*);
void proc7(int, int , int*);

int LOOPS = 5000000;

int i;
int b;
char c1;
char c2;
Record *ptr;
Record *ptr_next;

int _main() {
    proc0();
}

void proc0() {
    int il1, il2, il3;
    char cl, ci;
    int el;
    Record r1, r2;
    ptr_next = &r1;
    ptr = &r2;
    ptr->ptr = ptr_next;
    ptr->d = 1;
    ptr->e = 3;
    ptr->i = 40;
    //strcpy(ptr->str, "DRHYSTON...OME STRING");

    int idx;
    for(idx = 0; idx < LOOPS; idx++) {
        proc5();
        proc4();
        il1 = 2;
        il2 = 3;
        //strcpy(sl2, "DRHY... 2'ND STRING");
        el = 2;
//        b = ! func2(sl1, sl2);
        while(il1 < il2) {
            il3 = 5 * il1 - il2;
            proc7(il1, il2, &il3);
            ++il1;
        }
        proc1(ptr);
        for(ci = 'A'; ci <= c2; ++ci) {
            if(el == func1(ci, 'C')) {
                proc6(1, &el);
            }
        }
        il3 = il2 * il1;
        il2 = il3 / il1;
        il2 = 7 * (il3 - il2) - il1;
        proc2(&il1);
    }
}

void proc1(Record *in) {
    memcpy(in->ptr, ptr, sizeof(Record));

    in->i = 5;
    in->ptr->i = in->i;
    in->ptr->ptr = in->ptr;

    proc3(&in->ptr->ptr);
    if(in->ptr->d == 1) {
        in->ptr->i = 6;
        proc6(in->e, &in->ptr->e);
        in->ptr->ptr = ptr->ptr;
        proc7(in->ptr->i, 10, &in->ptr->i);
    } else {
        in->d = in->ptr->d;
        in->e = in->ptr->e;
        in->i = in->ptr->i;
        //strcpy(in->str, in->ptr->str);
        in->ptr = in->ptr->ptr;
    }
}

void proc2(int *x) {
    int il;
    int el;

    il = *x + 10;
    for(;;) {
        if(c1 == 'A') {
            --il;
            *x = il - i;
            el = 1;
        }
        if(el == 1) {
            break;
        }
    }
}

void proc3(Record **out) {
    if(ptr != 0) {
        *out = ptr->ptr;
    } else {
```

```
        i = 100;
    }
    proc7(10, i, &ptr->i);
}

void proc4() {
    int bl;

    bl = (c1 == 'A');
    bl |= b;
    c2 = 'B';
}

void proc5() {
    c1 = 'A';
    b = 0;
}

void proc6(int in, int *out) {
    *out = in;
    if(! func3(in) ) {
        *out = 4;
    }
    switch(in) {
        case 1: *out = 1; break;
        case 2: if (i > 100) {*out = 1;}
                else {*out = 4;} break;
        case 3: *out = 2; break;
        case 4: break;
        case 5: *out = 3; break;
        default:break;
    }
}

void proc7(int in1, int in2, int *out) {
    int il;

    il = in1 + 2;
    *out = in2 + il;
}

int func1(char in1, char in2) {
    char cl1;
    char cl2;

    cl1 = in1;
    cl2 = cl1;
    if(cl2 != in2) {
        return 1;
    } else {
        return 2;
    }
}
int func3(int in) {
    int el;

    el = in;
    if(el == 3) return 1;
    return 0;
}
```

C.2. Horizon version

```
class Record {
      variable ptr : Record = null
      variable d   : nativeint
      variable e   : nativeint
      variable i   : nativeint
      ;variable str : System.Array<'a => i8>
}

class Globals {
      variable i : nativeint
      variable b_ : nativeint
      variable c1 : nativeint
      variable c2 : nativeint
      variable ptr : Record
      variable ptr_next : Record
}

function () _main() {
         par {;A small script inserts calls here.
         ###marker###
         }
         ret
}

function () proc0(%n : nativeint) {
         alloc %g : Globals

         ; ptr->ptr = ptr_next
         load %ptr_next, %g.ptr_next
         load %ptr, %g.ptr
         store %ptr.ptr, %ptr_next

         ; ptr->d = 1;
         mov %tmp, 1
         store %ptr.d, %tmp
         ; ptr->e = 3;
         mov %tmp, 3
         store %ptr.e, %tmp
         ; ptr->i = 40;
         mov %tmp, 40
         store %ptr.i, %tmp

         mov %idx, 0
loop:    call %c *%idx.cmpeq %n
         bc %c, exit

         ; proc5();
         call proc5 %g
         ; proc4();
         call proc4 %g
         ; il1 = 2;
         mov %il1, 2
         ; il2 = 3;
         mov %il2, 3

         ; el = 2;
         mov %el, 2

         ;;; b = ! func2(sl1, sl2)

         ; while (il1 < il2) {
loop2:   call %c *%il1.cmpge %il2
         bc %c, after_while

         ; il3 = 5 * il1 - il2
         call %il3 *%il1.mul 5
         call %il3 *%il3.sub %il2
         ; proc7(il1, il2, &il3)
         call %il3 proc7 %g, %il1, %il2
         ; ++il1
         call %il1 *%il1.add 1

         ; }
         b loop2

after_while:
         load %ptr, %g.ptr
         ; proc1(ptr)
         call proc1 %g, %ptr
```

```
        ; for(ci = 'A'; ci <= c2; ++ci) {
        mov %ci, 65 ; 'A'
loop3:  load %c2, %g.c2
        call %c *%ci.cmpgt %c2
        bc %c, after_loop3

        ; if(el == func1(ci, 'C')) {
        call %tmp func1 %g, %ci, 67 ; 'C'
        call %c *%el.cmpne %tmp
        bc %c, ne

        call %el proc6 %g, 1
        ; }
ne:
        call %ci *%ci.add 1
        ; }
        b loop3

after_loop3:
        ; il3 = il2 * il1;
        call %il3 *%il2.mul %il1
        ; il2 = il3 / il1;
        call %il2 *%il3.udiv %il1
        ; il2 = 7 * (il3 - il2) - il1;
        call %il2 *%il3.sub %il2
        call %il2 *%il2.mul 7
        call %il2 *%il2.sub %il1

        ; proc2(&il1)
        call %il1 proc2 %g, %il1

        call %idx *%idx.add 1
        ; }
        b loop

exit:   ret
}

function () proc1(%g : Globals, %in : Record) {
         load %in_ptr, %in.ptr
         load %g_ptr, %g.ptr

         ; memcpy(in->ptr, ptr, sizeof(Record))
         load %tmp_ptr, %g_ptr.ptr
         store %in_ptr.ptr, %tmp_ptr
         load %tmp, %g_ptr.d
         store %in_ptr.d, %tmp
         load %tmp, %g_ptr.e
         store %in_ptr.e, %tmp
         load %tmp, %g_ptr.i
         store %in_ptr.i, %tmp

         ; in->i = 5;
         mov %tmp, 5
         store %in.i, %tmp
         ; in->ptr->i = in->i
         load %tmp, %in.i
         store %in_ptr.i, %tmp
         ; in->ptr->ptr = in->ptr;
         store %in_ptr.ptr, %in_ptr

         ; proc3(&in->ptr->ptr)
         call %tmp_ptr proc3 %g, %in_ptr
         store %in_ptr.ptr, %tmp_ptr

         load %d, %in_ptr.d
         call %c *%d.cmpne 1
         ; if(in->ptr->d == 1) {
         bc %c, ne1

         ; in->ptr->i = 6;
         mov %tmp, 6
         store %in_ptr.i, %tmp

         ; proc6(in->e, &in->ptr->e);
         load %e, %in_ptr.e
         call %e proc6 %g, %e
         store %in_ptr.e, %e

         ; in->ptr->ptr = ptr->ptr
         load %tmp_ptr, %g_ptr.ptr
         store %in_ptr.ptr, %tmp_ptr

         ; proc7(in->ptr->i, 10, &in->ptr->i)
         load %i, %in_ptr.i
         call %i proc7 %g, %i, 10
         store %in_ptr.i, %i

         ; }
         b after
ne1:
        ; else {
        ; in->d = in->ptr->d;
        load %d, %in_ptr.d
        store %in.d, %d
        ; in->e = in->ptr->e;
        load %e, %in_ptr.e
        store %in.e, %e
        ; in->i = in->ptr->i;
        load %i, %in_ptr.i
        store %in.i, %i
        ;;; strcpy(in->str, in->ptr->str);
        ; in->ptr = in->ptr->ptr;
        load %tmp_ptr, %in_ptr.ptr
        store %in.ptr, %tmp_ptr

after:
        ret
}

function (nativeint) proc2(%g : Globals,
                           %x : nativeint) {
         ; il = x + 10
         call %il *%x.add 10

         load %c1, %g.c1

         ; for(;;) {
loop:    ; if(c1 == 'A') { ('A' = 65)
         call %c *%c1.cmpne 65
         bc %c, ne

         ; --il;
         call %il *%il.sub 1
         ; x = il - i;
         load %i, %g.i
         call %x *%il.sub %i
         ; el = 1;
         mov %el, 1
         ; }
ne:
        ; if(el == 1) {
        call %c *%el.cmpne 1
        bc %c, loop

        ret %x
}

function (Record) proc3(%g : Globals,
                        %in : Record) {
         ; if(ptr != 0) { ;;; removed, useless.
         ; *out = ptr->ptr;
         load %ptr, %g.ptr
         load %out, %ptr.ptr

         load %i, %g.i
         ; proc7(10, i, &ptr->i);
         call %tmp proc7 %g, 10, %i
         store %ptr.i, %tmp

         ret %in
```

```
}

function () proc4(%g : Globals) {
        ; bl = (c1 == 'A')
        load %c1, %g.c1
        call %c *%c1.cmpeq 65
        bc %c, eq

        mov %bl, 0
        b after
eq:     mov %bl, 1
after:
        ; bl |= b;
        load %b_, %g.b_
        call %bl *%bl.or %b_
        ; c2 = 'B' ('B' = 66)
        mov %tmp, 66
        store %g.c2, %tmp
        ret
}

function () proc5(%g : Globals) {
        ; c1 = 'A' ('A' = 65)
        mov %tmp, 65
        store %g.c1, %tmp
        ; b = 0
        mov %tmp, 0
        store %g.b_, %tmp
        ret
}

function (nativeint) proc6(%g : Globals,
                           %in : nativeint) {
        ; out = in
        mov %out, %in

        ; if(! func3(in)) {
        call %c1 func3 %g, %in
        call %c, *%c1.cmpeq 1
        bc %c, no
        mov %out, 4
        ; }

no:     ; switch(in) {
        call %c *%in.cmpeq 1
        bc %c, case1
        call %c *%in.cmpeq 2
        bc %c, case2
        call %c *%in.cmpeq 3
        bc %c, case3
        call %c *%in.cmpeq 4
        bc %c, case4
        call %c *%in.cmpeq 4
        bc %c, case5
        ret %out

case1:  ret 1

case2:  ; if (i > 100)
        load %i, %g.i
        call %c *%i.cmpgt 100
        bc %c, over100

        ret 4
over100: ret 1

case3:  ret 2

case4:  ret %out
case5:  ret 3
}

function (nativeint) proc7(%g : Globals,
                           %in1 : nativeint,
                           %in2 : nativeint) {
        ; il = in1 + 2
        call %il *%in1.add 2
        ; out = in2 + il
        call %out *%in2.add %il
        ret %out

}

function (nativeint) func1(%g : Globals,
                           %in1 : nativeint,
                           %in2 : nativeint) {
        ; cl1 = in1
        mov %cl1, %in1
        ; cl2 = in2
        mov %cl2, %in2
        ; if(cl2 != in2) {
        call %c *%cl1.cmpeq %in2
        bc %c, else

        ; return 1
        ret 1
        ; return 2
else:   ret 2
}

function (nativeint) func3(%g : Globals,
                           %in : nativeint) {
        ; el = in
        mov %el, %in
        ; if(el == 3)
        call %c *%el.cmpne 3
        bc %c, else

        ; return 1
        ret 1
else:   ret 0
}
```

Bibliography

[1] Arstechnica commentary on htt. http://arstechnica.com/old/content/2002/10/hyperthreading.ars.

[2] C version of dhrystone. http://www.netlib.org/benchmark/dhry-c.

[3] Clang official website, features list.

[4] Context switching : Performance considerations. http://wiki.osdev.org/Context_Switching#Performance_Considerations.

[5] Cosmos offical website. http://www.gocosmos.org/.

[6] Dwarf debugging information format v3.0, pp139, section 7.6.

[7] From the inventors of unix system comes plan 9 from bell labs. http://www.lucent.com/press/0795/950718.bla.html.

[8] Gcc challenges. http://www.kaourantin.net/2006/09/gcc-challenges.html.

[9] Jamesm's kernel development tutorials. http://www.jamesmolloy.co.uk.

[10] Jnode official website. http://www.jnode.org/.

[11] Libpth. http://www.ossp.org/pkg/lib/pth/.

[12] Linux: C++ in the kernel? http://kerneltrap.org/node/2067.

[13] Llvm official website. http://llvm.org/.

[14] Sharpos official website. http://www.sharpos.org/.

[15] *UNIX System Programmer's Manual*.

[16] Wikipedia, ssa form. examples released without copyright.

[17] Apple. Apple technical brief on grand central dispatch. Technical report, Apple, 2009.

[18] K. Chandy and C. Kesselman. Compositional C++: compositional parallel programming. Languages and Compilers for Parallel Computing. In *Fifth International Workshop Proceedings. Springer-Verlag, New Haven, CT*, pages 124–44, 1992.

[19] Olivier Danvy and Andrzej Filinski. Representing control: A study of the cps transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[20] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 29–42, New York, NY, USA, 2006. ACM.

[21] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, and Dean M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17(5):12–19, 1997.

[22] M. Golm, M. Felser, C. Wawersich, and J. Kleinöder. The JX operating system. In *Proceedings of the USENIX Annual Technical Conference*, pages 45–58, 2002.

[23] John Goodacre and ARM. The design dilemma: Multiprocessing using multiprocessors and multithreading. http://www.techonline.com/learning/techpaper/193103846?_requestid=126672, June 2006.

[24] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[25] Intel. Intel pentium 4 processors 3ghz specification. Specification for pentium 4 chip with HTT.

[26] Intel. *64 and IA-32 Architectures Software Developer's Manual*, volume 3A: System Programming Guide Part 1, chapter 3.12, pages 3–51. 2007.

[27] Intel. Intel thread building blocks - reference manual. Technical report, Intel, 2009.

[28] Chris Jesshope and Bing Luo. Micro-threading: A new approach to future risc. In *ACAC '00: Proceedings*

of the 5th Australasian Computer Architecture Conference, page 34, Washington, DC, USA, 2000. IEEE Computer Society.

[29] Harry Kalogirou. Multithreaded game scripting with stackless python. http://harkal.sylphis3d.com/2005/08/10/multithreaded-game-scripting-with-stackless-python/.

[30] Donald Knuth. *Fundamental Algorithms*, chapter 1.4.2, pages 193–200. Addison-Wesley, 3 edition, 1997.

[31] T. Lindholm and F. Yellin. *Java™ Virtual Machine Specification*. Addison-Wesley Professional, second edition, 1999.

[32] P. Madany. JavaOS (tm): A Standalone Java Environment.

[33] B. Meyer. Systematic concurrent object-oriented programming. *Communications of the ACM*, 36(9):80, 1993.

[34] James Molloy and Matthew Iselin. The pedigree operating system. http://www.pedigree-project.org.

[35] Jörg Pfahler. Lightos. https://lightos.bountysource.com/.

[36] Microsoft Research. Singularity. http://research.microsoft.com/en-us/projects/singularity/.

[37] W. Stallings. *Operating Systems: Internals and Design Principles*, page 91. Prentice Hall, 5th edition, 2005.

[38] G.L. Steele and G.J. Sussman. *Scheme: An interpreter for extended lambda calculus*. Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1975.

[39] R.P. Weicker. Dhrystone: a synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, 1984.