BOSTON UNIVERSITY

GRADUATE SCHOOL OF ARTS AND SCIENCES

Dissertation

TERRIER: AN EMBEDDED OPERATING SYSTEM USING ADVANCED TYPES FOR SAFETY

by

MATTHEW DANISH

B.S., Carnegie-Mellon University, 2004

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2015

Approved by

First Reader

Hongwei Xi, Ph.D.
Associate Professor of Computer Science

Second Reader

Richard West, Ph.D.
Associate Professor of Computer Science

Third Reader

Chris Hawblitzel, Ph.D.
Research Scientist

Acknowledgments

My parents Bonnie and Keith Danish for everything; my primary advisor Hongwei Xi for his infinite patience with my questions; my frequent collaborator Richard West for supporting my foray into systems; my fellow students Likai Liu, Zhiqiang Ren, Ye Li and Mikhail Breslav; and the many other students, staff and faculty at Boston University who have helped me along in countless ways over the years. Without their support, none of this is possible.

TERRIER: AN EMBEDDED OPERATING SYSTEM USING ADVANCED TYPES FOR SAFETY

MATTHEW DANISH

Boston University, Graduate School of Arts and Sciences, 2015

Major Professor: Hongwei Xi, Ph.D.
Associate Professor of Computer Science

ABSTRACT

Operating systems software is fundamental to modern computer systems: all other applications are dependent upon the correct and timely provision of basic system services. At the same time, advances in programming languages and type theory have lead to the creation of functional programming languages with type systems that are designed to combine theorem proving with practical systems programming. The Terrier operating system project focuses on low-level systems programming in the context of a multi-core, real-time, embedded system, while taking advantage of a dependently typed programming language named ATS to improve reliability. Terrier is a new point in the design space for an operating system, one that leans heavily on an associated programming language, ATS, to provide safety that has traditionally been in the scope of hardware protection and kernel privilege. Terrier tries to have far fewer abstractions between program and hardware. The purpose of Terrier is to put programs as much in contact with the real hardware, real memory, and real timing constraints as possible, while still retaining the ability to multiplex programs and provide for a reasonable level of safety through static analysis.

Contents

List of Tables

List of Figures

List of Abbreviations

ACM	.....	Asynchronous Communication Mechanism
ARM	.....	Acorn RISC Machine
ATS	.....	Applied Type System
CPU	.....	Central Processing Unit
CSP#	.....	Communicating Sequential Programs–Sharp
EHCI	.....	Enhanced Host Controller Interface
GHC	.....	Glasgow Haskell Compiler
ICMP	.....	Internet Control Message Protocol
I/O	.....	Input/Output
IP	.....	Internet Protocol
IRQ	.....	Interrupt Request
μIP	.....	Micro IP
PAT	.....	Process Analysis Toolkit
PCI	.....	Peripheral Component Interconnect
NIC	.....	Network Interface Card
QH	.....	Queue Head
RISC	.....	Reduced Instruction Set Computing
RMS	.....	Rate Monotonic Scheduling
TCP	.....	Transmission Control Protocol
TD	.....	Transfer Descriptor
URB	.....	USB Request Block
USB	.....	Universal Serial Bus

Chapter 1

Introduction

Recent years have seen a proliferation of small, embedded, electronic devices controlled by computer processors as powerful as the ARM®. These devices are now responsible for tasks as varied as flying a plane, talking on a cellphone, or helping to perform surgery. Some of these tasks have severe consequences for a mistake caused by faulty programming or missed deadlines. The best defense against these mistakes is to prevent them from happening in the first place.

The operating systems software is fundamental to modern computer systems, such as these embedded devices: all other applications are dependent upon the correct and timely provision of basic system services. Ideally, the operating system is written with maximum attention to detail and the use of optimal algorithms. In practice, many difficult decisions must be made during the design and implementation of a realistic system.

There are many aspects of operating system development that contribute to this situation: the low-level behavior of hardware can be finicky, the asynchronous combination of system processes may produce unforeseen results, many of the resource-management problems are intractable to solve optimally, the slightest mistake can have profound consequences, and there is little room for any wasteful overhead. To maximize performance and ease of hardware interaction, most operating systems software is written in type-unsafe, low-level memory model programming languages like C or C++. But, this often leads to compromised reliability and safety because of programmer error.

At the same time, advances in programming languages and type theory have lead to the creation of functional programming languages with type systems that are de-

signed to combine theorem proving with practical systems programming. This allows programmers to bring the rigor of mathematical verification to important properties of their processes. I argue that the usage of these kinds of languages, in operating system development, can lead to better assurance that the processes running on an embedded device are correct, responsive, and safe.

The Terrier operating system project focuses on low-level systems programming in the context of a real-time embedded system, while taking advantage of a dependently typed programming language named ATS (Xi, 2004) to improve reliability. The purpose of this project is to identify effective, practical means to create safer, more reliable systems through use of advanced type system features in programming languages. I am also interested in the implications of having expressive programming language tools available, and the effects on plausible system design. For example, the Terrier OS moves much of the responsibility for program safety back out onto the programs themselves, rather than relying strictly on run-time checks or hardware protection mechanisms. For another, the Terrier program model is one in which asynchronous events play a central role in program design. These two shifts in thinking put more burden on the programmer—a burden that will be lightened through language-level assistance—but they also open up more flexibility in potential program design that will enable higher performance, better responsiveness and more naturally-written code in difficult problem domains.

ATS is a programming language with the goal of bringing together formal specification and practical programming. The core of ATS is an ML-like functional programming language that is compiled into C. The type system of ATS combines dependent and linear types to permit sophisticated reasoning about program behavior and the safety of resource usage. The design of ATS provides close coupling of type-safe functional code and low-level C code, allowing the programmer to decide the balance between specification and speed. The ATS compiler can generate code that does not

require garbage collection nor any other special run-time support, making it suitable for bare metal programming.

Using ATS, I have generated C code that links into my kernel to provide several critical components. I also encourage the use of ATS to help ensure the safety and correctness of programs that run under the OS. For example, programs that wish to communicate with one another are provided with libraries written in ATS that implement protocols that have been statically checked for safety and correctness. Most of these protocols fall into the category of asynchronous communication mechanisms, which ties into the central role that asynchronous event handling plays in the Terrier OS.

1.1 Operating system design

There are several competing goals of operating system design. For example, general purpose operating systems like Linux (Torvalds et al., 2014) might strive for maximum flexibility and practicality. Those systems are used by a wide diversity of people with many different applications. Figure 1.1 compares the goals of a general purpose operating system with those of a real-time operating system. In the latter case, responsiveness and predictability may be held as the most important properties of all.

But in doing so, the real-time operating system becomes less flexible, because it makes extraordinary demands of programs. It may ask for "Worst Case Execution Time" (Wilhelm et al., 2008) profiles, strict static scheduling parameters (Liu and Layland, 1973), priority ceiling protocol cooperation (Sha et al., 1990), or other kinds of information that require extensive analysis. A real-time system is not necessarily a high-performance system either. Some techniques used for high-performance, such as caching, are inherently difficult to predict (Basumallick and Nilsen, 1994) and therefore may be eschewed.

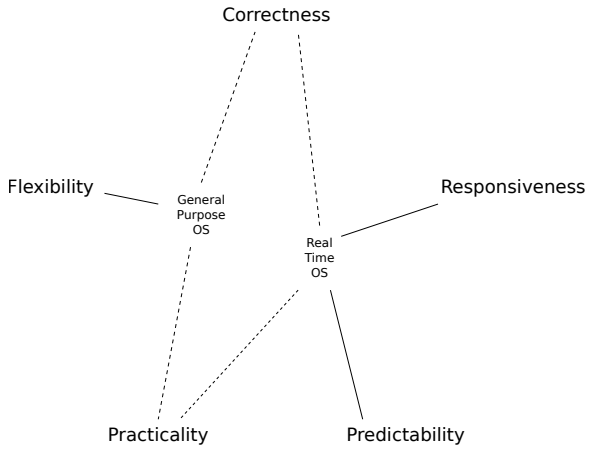

Figure 1.1: Goals of operating system designs

Either type of operating system might try to incorporate correctness guarantees in some form. Depending on how strong the assurances are, the amount of effort required could easily skyrocket. Or it could detract from other goals, such as flexibility, for instance, by making it more difficult to support diverse applications or hardware. It could hurt performance, practicality and responsiveness by requiring the use of run-time checks in certain cases. Compromise is necessary to balance these goals.

1.1.1 Programming languages and operating systems

In the same way, compromise is required in the design of programming languages and type systems. Also, many operating systems are intertwined with a particular programming language. In no particular order:

- Unix: C (Ritchie and Thompson, 1974)
- SPIN: Modula-3 (Bershad et al., 1995)
- Singularity: Sing# (Hunt and Larus, 2007)
- seL4: Isabelle, Haskell and C (Klein et al., 2009)
- House: Haskell (Hallgren et al., 2005)

It is sometimes said that types are "partial specifications" for programs. That means that the correctness guarantees of an operating system can be linked to the capability of the type system of the programming language in which it is written.

Some systems use a more powerful type system than others. For example, House is written in Haskell, a reasonably powerful programming language and type system, but it is supported by lifting the run-time environment straight from GHC (Peyton-Jones et al., 2014) and grafting it onto a bare-metal support framework. This means House must defer interrupt handling until known safe points in order to avoid breaking the garbage collector. Another popular system is Unix and C, which has a weak type system, but is popular with performance-oriented programmers.

The system that is being introduced in this text, Terrier, is intended to be a practical, incremental approach to leveraging advanced types while retaining efficiency enough to make it usable for real-time applications.

1.2 Motivation and structure

The purpose of most operating systems is to multiplex the hardware, to allow it to be shared between multiple processes and possibly multiple users. The most customary approach to this task is for the kernel to build environments that are an abstraction of the real hardware.

For example, the physical memory address space of most architectures is limited by real constraints, and is often strewn with many kinds of exceptional uses and alternate meanings. On the typical ARM-based architecture, general-purpose memory is restricted to certain ranges of physical addresses, while much of the rest is used for memory-mapped device communication. However, the details of physical addressing are almost always irrelevant to application developers, who just need a place to store their processes and data. So, typically, an operating system devotes significant effort to the construction of the "virtual memory" abstraction: it is the illusion of

flat, nearly unlimited, unrestricted and unconflicted memory address space that is provided to each process separately. The traditional operating system uses memory management hardware to give every process its own private address space, which is both a (virtual) resource, and a protection against unauthorized access into other processes' data.

Another way that operating systems abstract a real resource is through scheduling: the CPU can essentially only be working on a single process at a time, but there are many processes that want to make forward progress. Even in computers with multiple CPUs available, there are usually more processes than available processors. The standard way to deal with this problem is to time-share the CPU: each process in turn gets a chance to run, and then when that time-slice elapses, it is put to sleep and the next process is woken up. The exact details of which process is selected and for how long it runs is up to the specifics of the scheduling algorithm. But the common characteristic is that the CPU is rapidly switching between processes and that processes spend significant amounts of real time not running. However, that is not the abstraction that is provided to the application programmer. The traditional kernel is engineered to provide the illusion of continuous process execution that is uninterrupted unless explicitly requested by the application programmer.

There are other abstractions as well: filesystems that organize raw disk sectors into virtual resources such as files and directories, network stacks that multiplex the frame transceiving capabilities of network cards and do high-level routing of data, and a menagerie of other device drivers that make the computer usable in practice. Many of these abstractions are created to help multiplex but also to help protect processes from each other.

A great deal of the work done by the traditional operating system is to create the illusion of nearly limitless space and time for each process to enjoy separately. When application programmers do not care about real time, or real memory constraints,

then this approach makes sense. But does it still make sense for real-time applications? Does it make sense to go through the trouble of creating these abstractions and then breaking them for the sake of processes that must have a firm relation to real resources? Does it make sense to put up hardware barriers between two processes when the failure of one means the failure of the other, anyway?

Consider the case of a watchdog process running on an embedded system that is helping to fly an aircraft. It runs at precise periodic intervals to check on the status of the control systems. The programmers of this kind of embedded system have extensively studied the exact behavior of their hardware, have analyzed the worst-case execution time of their various pieces of software, and have spent a tremendous amount of effort to flush out all possible errors before deployment. There is no need for an abstraction of limitless time and space in any of the processes that are running in this system. Instead, what is needed is close tracking of real time, fast communication, minimal overhead, and rapid detection of hardware failures.

The watchdog process is, by nature, not continuously executing, but instead it is event-driven. It interacts with all other processes in the system, but it does not want to impose overhead on them, or any synchronization delays. Correctness of the program is important, but hardware memory protection would interfere with its mission. In a traditional operating system, this kind of program would probably be implemented as a kernel-level process that can see through the various abstractions of the kernel. That is one example of the limitations of a traditional operating system's model. But watchdog processes are hardly the only kind of program that may benefit from a less abstracted application model.

The underlying realization behind Terrier is that there already exists a model for applications that is more natural to the hardware: it is the model on which the kernel itself is built, the one provided by the hardware. Multiplexing is still needed, but, it ought to be possible to provide for that while also exposing an interface to processes

that most closely resembles the hardware. Thus, the philosophy behind the design for Terrier's program model:

- Asynchronicity and interrupts
- Explicit memory mapping
- Safety based on programming language features

and some of the derived features:

- Preemptive application-level scheduling
- Delegation of device control
- Flexible, asynchronous interprocess communication
- Ability to operate with or without hardware memory protection

1.2.1 Processes, programs and tasks

This text will use the term "program" to refer to a high-level language file, as well as the idea of a set of machine instructions packaged together as a single unit, intended to be loaded and executed by the kernel. The term "program" can be used to either mean the abstract concept of a program, at a high-level, or the concrete manifestation of said concept: as a string of binary code stored in a file or loaded into memory. The term "process" is similar to program but is typically used in contexts where specific operation of the kernel and the machine is being discussed. Finally, when the term "task" is used to describe a program, it is in a more abstract sense of something that requires CPU time and needs to be scheduled.

A "program model" is the environment and functionality provided by the kernel within which a program must exist and be constructed. A "program point" can either refer to a specific machine instruction within the compiled program, or to location

in the high-level language file that corresponds to a programmer's understanding of a specific point in the execution of the program. A "process context" is the state of the processor, including registers, at a given program point.

The kernel loads and manages a set of processes, scheduling them according to their static specifications, as described by Section 2.4.6. The following section goes into more detail about how processes may be interrupted by hardware events, and the way in which control is returned.

1.2.2 Asynchronicity and interrupts

The Terrier program model allows for the unexpected, sudden transfer of control from the current program point to a designated entry point in the application code. All programs are expected to have program code that forms the *entry handler*, and it should be placed at the designated entry point. The purpose of this code is to handle asynchronous events and to make decisions about what actions to take next. Several common behaviors will be made available as pre-compiled static object files or libraries that may be linked into applications to provide entry handlers, but it will also be possible for an application programmer to custom-design one.

In order to allow rapid resumption of execution, upon interrupt, the kernel captures a snapshot of the interrupted process context (with one exception) and later makes it available to the entry handler. That context will take the form of an array containing the program status register and the sixteen registers of the ARM processor. Restoration of context can be done within a few assembly instructions on the ARM processor in this way.

The kernel will also make available a table of interrupt status bits that reflect the current state of the interrupt controller. This will enable entry handlers to quickly decide whether or not a particular interrupt is of interest.

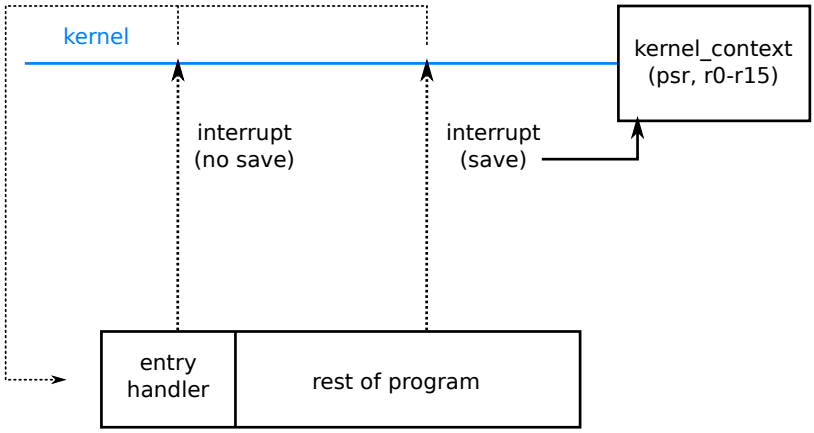

Figure 1.2: Entry handler and interrupts

Reentrancy and restartability

In order to avoid race conditions and make it possible for this mechanism to work without masking interrupts, the kernel reserves some special behavior for the entry handler. If a program is interrupted while the program counter is within the designated entry handler, as shown in figure 1.2, then the kernel will skip saving the interrupted process context, and will instead preserve the existing saved context. There are two main effects: one, the interruption of an entry handler does not destroy valuable program state before it can be examined, and two, the entry handler must be programmed to be entirely reentrant and restartable (Bershad et al., 1992) at any time. Entry handlers are further discussed in Chapter 2.4.1.

Although it is possible to program such an entry handler by hand, the use of verification techniques such as model-checking are well-suited for the task, and have been investigated, as discussed in Chapter 5.2.

Application control of scheduling

One of the consequences of the unusual entry handler design is that it allows a program to gain first-class access to its own contexts, or continuations, in a sense. Therefore, an entry handler can be designed that allows a choice between different process contexts,

or even the creation of an abstraction such as "user-level threads" but with support for preemption at any point, unlike typical implementations. This allows an application to manage scheduling within itself if the programmer chooses to use it, for example, to create user-level scheduling hierarchies (Parmer and West, 2008).

1.2.3 Explicit memory mapping

The ARM architecture makes hardware registers available through memory-mapped address space regions. Therefore, application programs will also be able to request explicit memory mappings from the kernel, along with desired attributes, in order to interact with hardware or provide interfaces. The current design has applications compiled with special sections that statically describe the necessary mappings, so that the kernel can analyze and decide which ones to fulfill and how. Dynamic mappings are also anticipated in the future.

Delegation of device control

Like a microkernel, most device support is expected to be fulfilled by application programs. Unlike a microkernel, these application programs are given very close control over the hardware. Using explicit memory mapping and access to the interrupt status table, device drivers may be written with minimal kernel interaction.

Interprocess communication

Communication between programs is mediated by shared memory and therefore falls under the explicit memory mapping regime. In the current design, applications that wish to communicate are expected to statically describe the name of the channel and the method of communication used. The kernel finds pairs of programs, then allocates the necessary physical memory, and creates virtual mappings if necessary, as shown in Figure 1.3. The actual communication between programs is expected to be handled through application libraries, written in ATS, which implement asynchronous

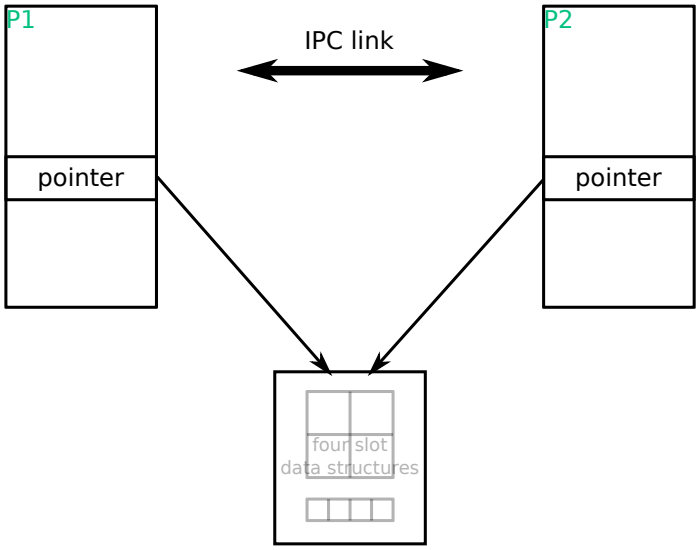

Figure 1.3: Example of interprocess communication

communication mechanisms, without kernel involvement.

The use of asynchronous communication mechanisms avoids the creation of dependencies between two schedulable entities, which avoids the pitfalls of synchronous IPC on fixed-priority real-time scheduling algorithms (Steinberg et al., 2005).

Hardware memory protection independence

Hardware memory protection is useful when dealing with unknown applications or during development to find and protect against undesired behaviors. However, it is less useful when all application code is known, or has been checked for safety with other means, and it does impose overhead on performance. Therefore, Terrier has been programmed from the ground up to operate correctly with or without hardware memory protection. It can be disabled prior to boot, currently as a compilation option.

The application programming model is affected by the use or non-use of virtual memory. In a virtual memory environment, all applications are provided the illusion of having the entire memory space to themselves. In a physical memory environment,

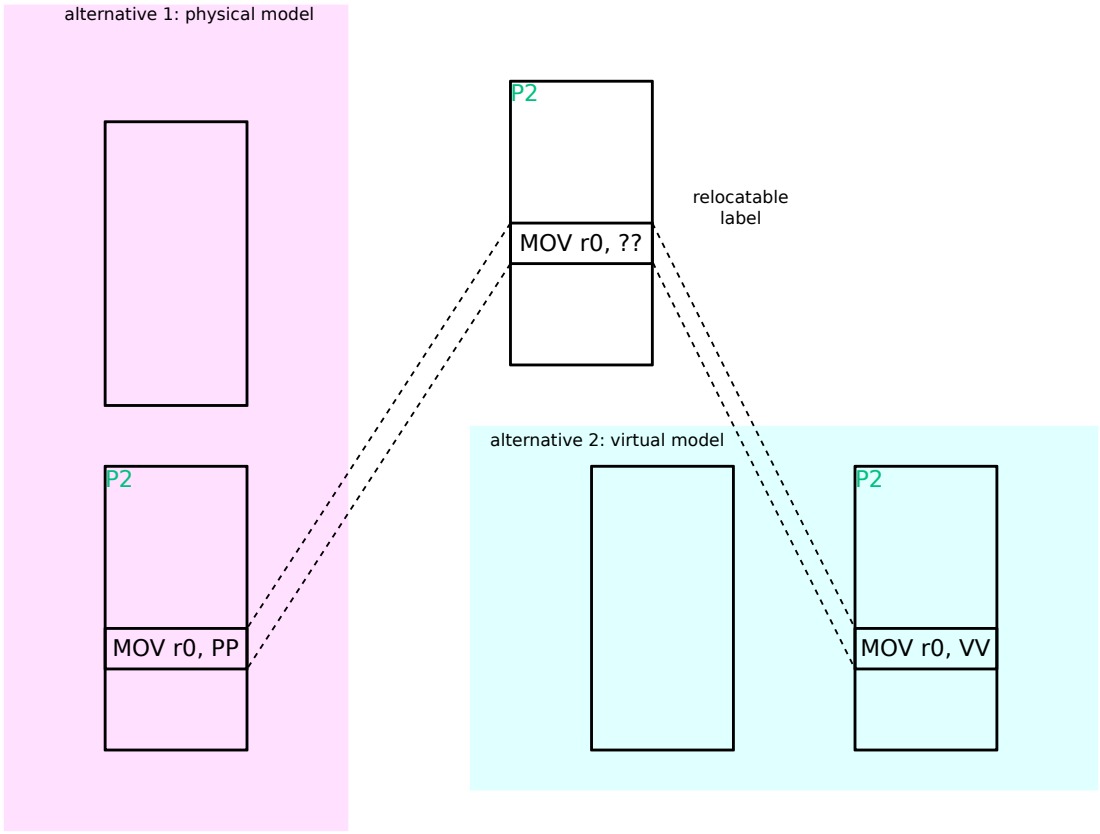

Figure 1.4: Hardware memory protection independence

applications are given different segments of memory for their own use and must avoid access outside of their space.

Terrier achieves the flexibility by requiring that all application programs be compiled with relocation sections enabled. Then, at load-time, Terrier rewrites the application binary code so that it fits into either one of the two memory models as shown in Figure 1.4.

1.2.4 Programming language-based safety

Terrier shifts much of the responsibility for program safety away from the hardware and kernel onto the application programmer. It also provides a more challenging model for programs. To make up for that, Terrier encourages the use of the ATS func-

	Proposition	Type
Linear	`view`	`viewtype`
Non-linear	`prop`	`type`

Table 1.1: ATS terminology

tional programming language, which allows the use of dependent and linear types, while allowing the programmer to avoid most or all of the overhead typically associated with high level languages.

Types in ATS

ATS has multiple levels of types (i.e. "types" of types) that are necessary in order to have usable, complex types that can describe interesting properties. ATS uses the term *sort* (Xi, 2004) to distinguish "types" of types from types of values. The programmer may quantify at the sort-level in order to describe particular values to index a type. Alternatively, you might say, these are the values upon which a type is "dependent."

ATS also has other dichotomies: proposition vs "real" type, linear vs non-linear. Linear types are also nicknamed *views*, from the common usage of protecting pointers that have a particular "view" of memory. The various ATS names for these concepts are shown in table 1.1.

In the example of figure 1.5 the `@` (pronounced: at-view) is a constructor for a view, and it has two parameters: the type of value found in memory (`(int, char)` in the example), and the address at which it is found (`L` in the example). The at-view

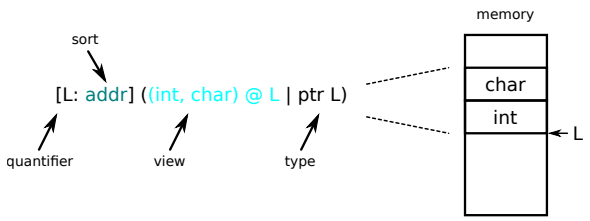

Figure 1.5: Views and types in ATS syntax

abstractly represents the "view" of that slice of memory found at that address.

Resource management

One of the biggest tasks in system development is that of resource management. As a result, one of the biggest source of bugs in systems is caused by resource mismanagement. The linear propositions and types provided by ATS are well suited to addressing this problem. In short, a value of a linear type is consumed once and exactly once, avoiding the common errors of memory leak or dangling pointer. In addition, the dependent types of ATS are useful for describing exactly the kind of details that need to be statically checked for safe usage. For example, array bounds length, or contents of a memory region accessible through a particular pointer.

Synchronization

Another use of linear types and properties is for synchronization. For example, locks can be considered a kind of resource that must be acquired and released. Linear types can be used to enforce sequencing of steps; they are also useful for locks that must be acquired in a particular order to avoid deadlocking.

Avoiding overhead

After compilation, all of the types are stripped away and therefore none of it affects runtime performance.

And even though ATS is a high level language, it primarily works with "flat types" that are essentially equivalent to C's data representation. They are types of varying sizes, often referred to as "unboxed types" in other contexts. For example, the low-level implementation of a flat "pair" is just a C struct with two members.

To go along with that, ATS has support for templates as a means of gaining the practical benefits of parametric polymorphism. Like C++, the templates of ATS are generated for each different type, and the names are mangled to co-exist with each

other.

1.2.5 Scheduling

In order to provide guaranteed partitions of processor time to each program, Terrier uses a static priority, rate monotonic (Liu and Layland, 1973) real-time scheduler algorithm. The design shies away from traditional blocking schemes that use systems of queues to wake-up programs when their request is available. Instead, the system programmer is expected to choose parameters for the scheduler that give each program an adequate amount of time to operate, at periodic intervals. Each program will respond to events in turn, when given control of the CPU. Most communication is expected to be conducted asynchronously, so programs will continue to run and execute code after invoking a communication protocol. Programs that must run with significant concurrency to each other are expected to be scheduled on separate CPUs, decoupled by asynchronous communication mechanisms, which helps avoid unnecessary delays.

1.3 Related work

Verification of operating system software has been a goal of many researchers and programmers over the decades. In a way, it is reminiscent of the effort by early 20th century mathematicians to find a firm basis for their theories rooted in purely logical reasoning. Since the operating system forms the basis of a computer system, its correctness is vital to the correct operation of all other software running on the system. However there are many challenges: the operating system must manage many different layers of hardware and software together, many different applications and devices that may be independently designed, and it must accomplish all of these feats within a fraction of a millisecond. So while the operating system is a kind of program, it is a very special kind of program that deserves its own consideration.

1.3.1 Program verification

There have been a great many efforts to improve the reliability of programs using programming language techniques. Some are applied to existing languages, like CCured (Necula et al., 2005), which checks memory safety of ordinary C programs by applying a carefully designed strong type system that differentiates between different kinds of pointer use. It inserts run-time checks where memory safety cannot be proven statically, and tracks meta-data about certain pointers in order to implement those run-time safety tests—these are often called "fat" pointers. In a slightly different approach, Cyclone (Jim et al., 2002) is a memory-safe dialect of C. It achieves its goal by restricting the behavior of C programs and then recovering some of that expressiveness by giving the programmer features that: insert NULL-pointer checks, use "fat" pointers when pointer arithmetic is desired, have growable regions as an alternative to classical manual memory management, and more.

TESLA (Anderson et al., 2014) is a tool that allows programmers to annotate their C code with temporal assertions that describe safety properties in a language based on Linear Temporal Logic. The assertions generate code that is used to instrument the program at run-time in order to check the given properties. TESLA is an entirely dynamic analysis tool and therefore imposes a run-time penalty on instrumented programs. In contrast, MUVI (Lu et al., 2007) automatically detects certain classes of concurrency bugs statically using pattern analysis on multi-variable access correlations. It is able to detect when correlated variables are not updated in a consistent way, and when correlated accesses are not protected in the same atomic section. Although useful for finding those two classes of bugs, MUVI cannot verify arbitrarily specified properties in general.

Other efforts have focused on the creation of novel programming languages. BitC (Sridhar et al., 2008) is a language intended to be used for systems programming with

an ML-style type system with effects. It allows precise control of the representation of types in memory while enjoying the advantage of static type inference. Theorem-proving is syntactically supported but left to a (yet to be created) plugin or third-party application.

Bedrock (Chlipala, 2011) is a Coq (Herbelin, 2009) library that forms a programming language to address these concerns. It uses a computational approach to verify low level code that allows programmers to express invariants using functional constructs already present in the Coq environment. Bedrock contains a generic machine language syntax for describing program implementations. There are few operations, including: describing registers, memory dereferencing, conditionals, and jumps. On top of this, assertions in separation logic allow a programmer to define pre- and post-conditions on functions. Given that ATS and C share the same data representation, it is possible that I could make use of the C code verified by Bedrock when constructing ATS programs.

Dafny (Leino, 2010) is an imperative programming language designed to support the static verification of programs using the Boogie intermediate language (Barnett et al., 2006) and the Z3 SMT solver (De Moura and Bjørner, 2008). The Dafny compiler produces code suitable for the .NET platform.

1.3.2 Operating system verification

With operating system software being fundamental to the correct functioning of the entire machine, many efforts have been directed at trying to verify operating systems. The seL4 project is based on a family of microkernels known as L4 (Klein et al., 2009). In that work, a refinement proof was completed that demonstrates the adherence of a high-performance C implementation to a generated executable specification, created from a prototype written in Haskell, and checked in the Isabelle (Paulson, 1994) theorem proving system. The prototype itself is checked against a high-level design.

One difference with my work is that I seek to eliminate the phase of manual translation from high to low level language. Another difference is that, while the seL4 approach can certainly bring many benefits, I feel that the cost associated with it is too high for ordinary use. For example, it may turn out to be intractably difficult to apply this technique to a multiprocessor kernel. That is currently an open problem according to Elphinstone and Heiser (2013).

Singularity (Hunt and Larus, 2007) is a microkernel OS written in a high-level and type-safe language that employs language properties and software isolation to guarantee memory safety and eliminate the need for hardware protection domains in many cases. In particular, it makes use of a form of linear types in optimizing communication channels. Singularity was an inspiration for Terrier, although several goals are different. For instance, Terrier seeks to avoid, as much as possible, the overhead associated with high-level languages. Terrier's design is more explicitly geared towards embedded devices responding to real-time events. And inter-program communication in Terrier is left open enough to accommodate multiple approaches, tailored to the particular application domain.

House (Hallgren et al., 2005) is an operating system project written primarily in the Haskell functional programming language. It takes advantage of a rewrite of the GHC (Peyton-Jones et al., 2014) run-time environment that eliminates the need for OS support, and instead operates directly on top of PS/2-compatible hardware. Then a foreign function interface is used to create a kernel written in Haskell. There is glue code written in C that glosses over some of the trickiness. For example, interrupts are handled by C code that sets flags that the Haskell code can poll at safe points. This avoids potentially corrupting the Haskell heap due to interruptions of the Haskell garbage collector while it is an inconsistent state. SPIN (Bershad et al., 1995) is a pioneering effort along these lines that used the Modula-3 language (Cardelli et al., 1989) to provide a protection model and extensibility. In general, these types of

systems do not tackle the problem of high-level language overhead, generally do not handle multiprocessing well if at all, and only offer guarantees as good as their type system can handle.

Verve (Yang and Hawblitzel, 2010) improves on this approach by splitting the system into two parts: a "Nucleus" written in verified assembly language that exports abstractions of the underlying hardware, and a kernel layer on top of that, which provides traditional services. Verve verifies the correctness of the Nucleus, but only verifies the safety of the kernel. Verve is fully mechanically verified for type safety, but it does not support multiple processors, uses a stop-the-world GC, and keeps interrupts disabled during collection, making it unsuitable for real-time applications. The Ironclad Apps framework (Hawblitzel et al., 2014) extends the Verve OS with some advanced features to support end-to-end security, secure hardware, and the Dafny language. Ironclad Apps uses a modified Dafny compiler that outputs BoogieX86 (Hawblitzel and Petrank, 2009) code that can be verified independently and easily translated into machine code.

Both VFiasco (Hohmuth and Tews, 2005) and Verisoft (Alkassar et al., 2008) take a completely different approach to system verification. Verisoft relies upon a custom hardware architecture that has itself been formally verified, and a verified compiler to that instruction set. VFiasco claims that it is better to write the kernel in an unsafe language such as C++ and then mechanically generate theorems from that source code, to be discharged by an external proof engine.

1.3.3 Asynchronous communication mechanisms

Simpson (1990) designed an ACM called the "four-slot mechanism" intended to allow the communication of a piece of reference data from a single writer to a single reader without the use of any mutual exclusion mechanisms nor special atomic operations. The name "four-slot" comes from the memory space requirement: the mechanism

requires the use of an array that can hold up to four copies of the reference data. The same paper also described several of the desired properties of such mechanisms, as well as a scheme for categorizing the behavior of certain ACMs. This "four-slot mechanism" was later generalized (Simpson, 1997a) to work with multiple readers and multiple writers, resulting in a "$2 \times nw \times (n_r + 1)$-slot mechanism", where n_w, n_r are the number of writers and readers respectively.

The ATS implementation (Danish and Xi, 2014) of Simpson's four-slot mechanism is compact, efficient and shows how expressive types can provide useful assurances at a low-level without intruding into run-time performance of critical code or requiring voluminous quantities of proof-writing.

Simpson (1992) also developed a model-checking technique called "role model analysis" and then applied it to his four-slot mechanism (Simpson, 1997b) to verify properties of coherency and freshness. Henderson and Paynter (2002) created a formal model of the four-slot mechanism in PVS and used it to show that it was atomic under certain assumptions about interleaving. Rushby (2002) used model-checking to verify coherency and freshness in the four-slot mechanism but also found the latter can only be shown if the control registers are assumed to be atomic. By comparison, my approach has been to encode pieces of the desired theorems into the type system, apply it to working code, and then allow the type-checker to verify consistency. If a mistake is made, it will be caught prior to compilation. Or, if the type-checker is satisfied, then the end result is efficient C code that may be compiled and linked and used directly by applications.

Subsequent to Simpson's work, Chen and Burns (1997) collaborated on a "three-slot mechanism" that offered the same features as the "four-slot mechanism" except it is able to operate with a smaller memory requirement. This mechanism was proven to be safe and effective as long as the atomic operation `Compare-And-Swap` is available from the hardware. This too was later generalized (Chen and Burns, 1998) into an

"$n+2$-slot mechanism" with a single writer and n readers, using the atomic operation `Test-And-Set`. Both Simpson's and Chen's ACMs have a rising memory requirement that scales with the number of programs communicating with each other.

The following year, Chen and Burns (1999) introduced one answer to the scaling space problem: a "timing-based" ACM based on circular buffers. In this approach, the size of the circular buffer is statically configured based on the static rate-monotonic scheduling parameters. For many real-time systems, this is a reasonable approach, although it does require re-analysis any time the static scheduling parameters are changed. By comparison, the fixed-slot mechanism remains in the category of "algorithm-based" solutions, since it does not require foreknowledge of scheduling parameters.

1.3.4 Related designs

First popularized in the 1980s, a microkernel design (Liedtke, 1995) keeps as much kernel functionality as possible out of the hardware-privileged level. Normally in such a design, privileged code is restricted to the minimum necessary to implement hardware memory protection, most basic scheduling, and interprocess communication. Higher level operating system functions are moved out into unprivileged processes or threads, and then granted access to the hardware on a restricted basis. These processes or threads are expected to all coordinate with each other, and with other programs, by using the core interprocess communication mechanisms.

Originally developed at MIT, an exokernel (Engler et al., 1995) is designed to have as few abstractions as possible between programs and the hardware. Therefore, exokernels are tiny, and mostly mediate between various "library operating systems," which are software layers that implement more traditional functionality. The exokernel's job is to multiplex access to the hardware, and to protect those "library operating systems" from each other. By leaving most of the real functionality out of

the kernel, the designers of exokernels hope to impose as little prejudice as possible in the design of programs. Instead, custom tailored "library operating systems" can be used to create a more conducive environment for each program separately, if need be.

Barrelfish (Baumann et al., 2009) introduced the model of a "multikernel": a design that treats multiple processors as being independent members of a distributed system, communicating explicitly through asynchronous messages. The motivation comes from the increasing number of processor cores readily available in computer systems: the abstraction of shared memory across many cores is becoming increasingly difficult to maintain efficiently. By moving towards a shared-nothing design, with explicit message passing between distributed components, the authors feel that they are designing an OS model that is more appropriate for upcoming generations of hardware.

Quest-V (Li et al., 2014) is a "separation kernel" (Rushby, 1981) that takes advantage of modern virtualization hardware on multicore systems to provide component isolation while avoiding much of the overhead of involving a central monitor or hypervisor. Shadow page table mappings implemented in hardware allow virtual machines to transparently run within virtual memory without hypervisor interaction; additionally, with modern hardware, it is possible to deliver interrupts directly to guest kernels without exiting the virtual machine. When each processor is mapped to a single guest, there is almost no reason at all to exit the virtual machine, unless the guest misbehaves. Therefore, under this scheme, each guest kernel can run at full speed while interacting with other guest kernels only through specially arranged shared memory regions. I collaborated with Ye Li and Richard West on this project prior to beginning work on Terrier.

1.4 Contributions

Terrier is a new point in the design space for an operating system, one that leans heavily on an associated programming language, ATS, to provide safety that has traditionally been in the scope of hardware protection and kernel privilege. The design takes the liberty to devolve many important functions into programs, but does not put up the protection barriers of typical microkernels. It also does not expect a large number of small programs communicating with extensive use of interprocess communication, but rather a more coarse-grained set of tightly knit subsystems that are then loosely coupled through asynchronous communication mechanisms. Like exokernels, Terrier tries to have far fewer abstractions between program and hardware. Unlike exokernels, Terrier expects typical application development to occur at this low-level program model, rather than being subsumed under "library operating systems," which themselves create abstractions. Terrier is also explicitly directed at supporting SMP, real-time applications, and uses a real-time scheduling algorithm rather than the typical round-robin method found in an exokernel (Deville et al., 2004). The purpose of Terrier is to put programs as much in contact with the real hardware, real memory, and real timing constraints as possible, while still retaining the ability to multiplex programs and provide for a reasonable level of safety through static analysis.

To itemize, the contributions of Terrier are:

- An operating system with a significant portion written in a high-level, functional programming language with advanced type system features such as dependent and linear types.
- A program model for the operating system that emphasizes the underlying machine model, putting programs as much in contact with real hardware, real memory, and real timing constraints as possible, while using advanced program-

ming language features to manage the additional complexity safely.

- A style of programming that applies programming with proofs in a gradual manner in order to debug code, and create an increasing amount of confidence in its correctness, while retaining the low-level efficiency necessary for system programming.
- A new asynchronous communication mechanism that can have multiple readers without increasing its memory space requirements, developed using this new style of programming, and useful for applications written in this operating system.

1.5 A crash course in the ATS programming language

1.5.1 From types to dependent types

In order to help you follow the examples in this text, I will introduce some of the features of ATS that are used. ATS is a functional programming language at heart, with a syntax based on the ML family of languages. For example, a simple function named `test` is defined in Listing 1.1. It takes a single parameter `x` of type `int`, and it returns the value of `x` plus 1, which is also of type `int`. The use of `let..in..end` is unnecessary in this instance but demonstrates a common syntactic structure in the language.

Although ATS supports a limited form of type inference that allows you to omit certain type annotations, it is best to put all of them on function definitions be-

```
fun test (x: int): int =
let
  val y = x + 1
in
  y
end
```

Listing 1.1: A simple function

```
fun test {i: int} (x: int (i)): int (i + 1) =
let
  val y = x + 1
in
  y
end
```

Listing 1.2: A simple function with dependent types

cause the truly advanced types supported by ATS are not compatible with full type inference.

Earlier, I briefly described the layers of the type system of ATS, in Section 1.2.4. The above example does not take advantage of dependent types, but we can modify it to do so, as seen in Listing 1.2.

Here, `{i: int}` defines a variable at the type-level of the sort named `int` (not to be confused with the type named `int`, based on context), while `int (i)` is the name of an indexed type intended to represent specifically the integer `i`. The fact that the function now has a return-type of `int (i + 1)` is significant: the function body must satisfy this type or else it will fail to compile. Luckily for us, the addition function has the type signature shown in Listing 1.3.

ATS has overloaded the `+` operator with this function and also recognizes that the constant value `1` has type `int (1)`. The type-checker is smart enough to automatically put together the pieces and conclude that the expression `x + 1` in the function `test` has the type `int (i + 1)` without further programmer intervention.

I realize that the sort, type and value distinction can be confusing. For now, think of variables defined within curly brackets `{...}` as being used for specifications and constraints on the types that are applied to the parameters of functions. Consider

```
fun add {i, j: int} (x: int (i), y: int (j)): int (i + j)
```

Listing 1.3: Type signature of indexed-`int` addition

```
fun lookup {ty: type} {i, n: nat} (
    idx: int (i),
    A: arrayref (ty, n)
  ): ty =
let
  val y = A[idx] // This causes a type error
in
  y
end
```

Listing 1.4: Dependently typed array access (broken)

the following example of how specifications can be useful for practical programming as shown in Listing 1.4, which indicates that the array access shown will lead to a type error at compile-time.

Why does that type error happen? Because in ATS, array access requires evidence that the index will not be outside of the array bounds. We can provide that evidence in a number of ways: for example, by employing a run-time check such as an `if`-statement that tests the value of the variable `idx` to be sure that it is small enough. But actually, that would not work in this case because there are no variables telling us the length of the array: the only knowledge we have of array length is found in the type-level variable `n` and that variable is erased by the compiler prior to run-time.

Therefore, we have another option: we can modify the definition of the type-level variable `i` to give us a precondition that statically enforces the safety condition that we need. Any callers of the function `lookup` will then have to satisfy that condition, somehow, before successfully passing the type-checker. The type-level variable `i` is introduced by the quantifier `{i, n: nat}`, which is similar to the previous example but using a sort that is named `nat` intended to model 0-based natural numbers. ATS has syntax to conveniently add conditions to these kinds of quantifiers in a manner that resembles mathematical set-notation: `{i, n: nat | i < n}` will do. With that annotation, the example will successfully pass the type-checker. Any callers to the function `lookup` will have to demonstrate that the `idx` argument is both a natural

```
fun lookup {ty: type} {i, n: nat | i < n} (
    idx: int (i),
    A: arrayref (ty, n)
  ): ty =
let
  val y = A[idx] // OK
in
  y
end
```

Listing 1.5: Dependently typed array access (fixed)

```
fun test (x: & int): void =
begin
  x := x + 1
end
```

Listing 1.6: A simple function using call-by-reference

number and less than the length of the array.

1.5.2 Call-by-reference

ATS is a functional language but it also supports a number of features that will endear it to C programmers. One of those features is fully integrated, built-in support for call-by-reference. Listing 1.6 defines our `test` function in a different way.

This function simply updates the `x` parameter in-place, a change that will be reflected within the calling function as well. Dependent types can be utilized this way as well, as shown in Listing 1.7.

But this will not work as is. The reason is that the type of `x` is specifically `int (i)` and that the value of `i` is not equal to `i + 1`. In order to correctly describe the specification of the function at this level of granularity, we need to be able to reflect

```
fun test {i: int} (x: & int (i)): void =
begin
  x := x + 1 // Type error
end
```

Listing 1.7: Call-by-reference and dependent types (broken)

```
fun test {i: int} (x: & int (i) >> int (i + 1)): void =
begin
  x := x + 1 // OK
end
```

Listing 1.8: Call-by-reference and dependent types (fixed)

the change to the parameter at the type level. And that is accomplished with this bit of syntax in Listing 1.8.

Now the function signature correctly describes the type of `x` upon entry to the function and upon exit from the function.

1.5.3 Protecting resources with linear types

In a previous example, Listing 1.5, I showed how you can use dependent types to check array-bounds statically. However, the array itself, with type `arrayref`, can have indefinite extent and therefore must have its memory managed automatically – most likely by a garbage collector. In ATS, however, we do not require the use of automatic memory management techniques such as that. There are many cases where garbage collection is undesirable or impractical, in the kind of low-level system programming for which ATS is intended. In fact, ATS offers a whole collection of types for which proper resource management is a requirement imposed by the type-checker. As laid out in Table 1.1, ATS offers both Linear Propositions (known as "views") and Linear Types (known as "viewtypes"). The introduction and elimination rules for these views and viewtypes are based on those for linear substructural logic. As a practical matter, the implication is that values of a linear type must be consumed once and only once, eventually.

One of the most immediately apparent uses for linear types is to ensure that allocated memory is freed and that dangling pointers are not dereferenced. The invocation of an "allocation" function using linear types creates a statically-checkable obligation to show that all control paths result in the eventual "release" of the allo-

```
absviewtype my_resource // Introduce an abstract linear type

// Several function signatures (implementations not shown)
extern fun allocate_my_resource (): my_resource
extern fun use_my_resource (my_resource): my_resource
extern fun release_my_resource (my_resource): void

fun test (): void =
let
  val x1 = allocate_my_resource ()
  val x2 = use_my_resource (x1)
in
  release_my_resource (x2);
  use_my_resource (x2) // Type error: x2 is not available
end
```

Listing 1.9: A basic use of linear types

cated resource. Furthermore, once the resource is released, any attempt to try and use it again results in a type error.

In Listing 1.9, `allocate_my_resource` is said to "produce" a value of viewtype `my_resource`, while `release_my_resource` is said to "consume" a value of that viewtype. The function `use_my_resource` therefore does both: it consumes and then produces a value of that viewtype.

Functions such as `use_my_resource` are so common that ATS has introduced a much more convenient syntax for achieving the same consumption-and-reproduction signature. By annotating a linear type with the bang prefix (`!`) you inform the type-checker that the value should be reproduced after the function returns, as shown in Listing 1.10.

The program flows a lot more naturally now, in a way that might even be familiar to a C programmer, but with all the guarantees of the linear type-checker still enforced.

```
absviewtype my_resource

extern fun allocate_my_resource (): my_resource
extern fun use_my_resource (! my_resource): void
extern fun release_my_resource (my_resource): void

fun test (): void =
let
  val x = allocate_my_resource ()
in
  use_my_resource (x); // OK
  release_my_resource (x);
  use_my_resource (x) // Type error: x is not available
end
```

Listing 1.10: Consumption and reproduction, conveniently

1.5.4 Linear and dependent types

Naturally, we will want to have the advantages of both linear and dependent types when writing complex low-level system software. One of the most frequent use cases for such types is with array manipulation.

In Listing 1.11 you can see an example of a linear proposition parameter named `pf_A` being used to provide evidence that an array of length `n` exists at address `l`. I haven't shown explicit use of propositions until now, but you can take them as being similar to types, but having no run-time presence or effect. Any parameter to the left side of the vertical bar symbol `|` is intended to be a purely static argument, of interest to the type-checker, but erased prior to run-time. Only the parameters that come after the vertical bar `|` will be part of the final, compiled program: in this case,

```
fun test {n: nat} {l: addr} (
    pf_A: ! array_v (int, l, n) |
    A: ptr l,
    len: int n
  ): int =
if len > 0 then A[0] else 0
```

Listing 1.11: Linear arrays

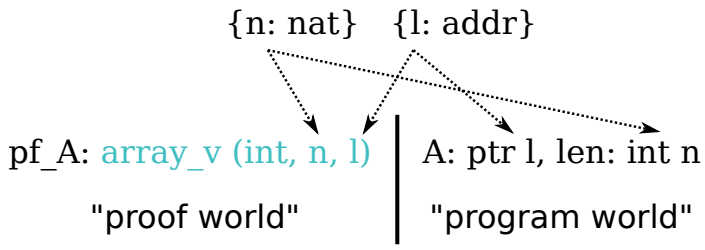

Figure 1.6: Type indices link proof and program

a pointer and an `int` value.

The expression `A[0]` transforms into an array access at index 0, which requires evidence that `A` is an array, that the array has at least one element, and that the type of the element matches the specified return type here (which is `int`). The linear proposition (known as a "view") named by `pf_A` provides the evidence that `A` is an array because both `pf_A` and `A` are indexed by the same static variable `l`. As shown in Figure 1.6, these type and proposition indices are what provide the link between the so-called "proof world" and the "program world" so that compile-time proofs are able to make useful, logical, and checkable statements about program behavior at run-time. Similarly, there is a static variable `n` that links the length of the array view to the value of the `int` parameter named `len`. Thus, the ATS type-checker is able to deduce that an `if`-statement testing that the value of `len > 0` is sufficient evidence that the array has at least one element.

1.5.5 Linear types that evolve

When writing complex code using linear types, you may come across a situation where you consume a resource and then reproduce it – but in a slightly different form. A classic example may be the `realloc` function from the C standard library. One example of what would that look like in ATS is shown in Listing 1.12.

For simplicity, I will assume that we are only dealing with an array of `int`s that needs to be resized. The function `realloc` requires a view that proves you have an

```
int *realloc (int *ptr, size_t size); // in C

extern fun realloc {n, n': nat} {l: addr} (
    array_v (int, l, n) |
    ptr l,
    size_t n'
  ): [l': addr] (array_v (int, l', n') | ptr l')
```

Listing 1.12: `realloc` in ATS

array, as well as the array itself, and the new size. It consumes the old view and returns a new view of an array of the new size, as well as a possibly-changed pointer. ATS will not allow us to assume that the old address named by `l` is equal to the new address named by `l'`. The new address has been introduced by the existential quantifier `[l': addr]`. All it says is that there exists an address, and it binds it to a static variable name.

Oftentimes, though, this is much more verbose than we'd like. So ATS provides a syntax using the bang prefix (`!`) that simplifies this pattern. That syntax is shown in Listing 1.13.

The combination of the bang prefix (`!`) and the shift-right operator (`>>`) is intended to be similar in functionality to the call-by-reference feature discussed in Section 1.5.2. Except, of course, since this is merely a view, it does not appear in the compiled output, and therefore calling convention is irrelevant. The annotation is purely for the sake of the type-checker. Another annotation not to be missed is the hash-tag (`#`) that must now be used in combination with existential quantification: if you want to refer to such existentially quantified variables in the scope after the shift-right

```
extern fun realloc {n, n': nat} {l: addr} (
    ! array_v (int, l, n) >> array_v (int, l', n') |
    ptr l,
    size_t n'
  ): #[l': addr] ptr l'
```

Listing 1.13: `realloc` in ATS, evolved

(`>>`) operator then you must indicate that by putting a hash-tag (`#`) as prefix to the quantifier. That is all.

Even though this feature is commonly used with views, it is also valid to use with viewtypes: then the function call becomes similar, at type-checking time, to the call-by-reference notation that uses (`&`), but it retains call-by-value semantics operationally. This might be useful for examples like file handles: the handle does not change, but the state of the object behind it might. If you want to represent that state at the type-level, you might use call-by-value but with linear update, as shown in Listing 1.14.

What this tells me is that the actual value of `my_handle` does not change, but I wish to model something about its change of state at the type level.

1.5.6 Flat types

Another practical feature offered by ATS is the family of "flat types" (also known as unboxed types). These are useful for working with C types that may be of non-pointer size. Most high-level functional programming languages require that the run-time representation of values by uniformly sized. A common technique for handling larger-sized values is to allocate a piece of memory and then store a pointer in place of the value: this is known as "boxing" of the value. Because this allocated piece of memory is created behind the scenes, it must be automatically managed by techniques such as garbage collection. But in ATS we need to be able to program without garbage collection, and we need to be able to manipulate values that do not fit neatly into

```
absviewtype my_handle (state: int)

extern fun change_state {s: int} (
    ! my_handle (s) >> my_handle (s')
  ): #[s': int] void
```

Listing 1.14: A call-by-value viewtype that evolves

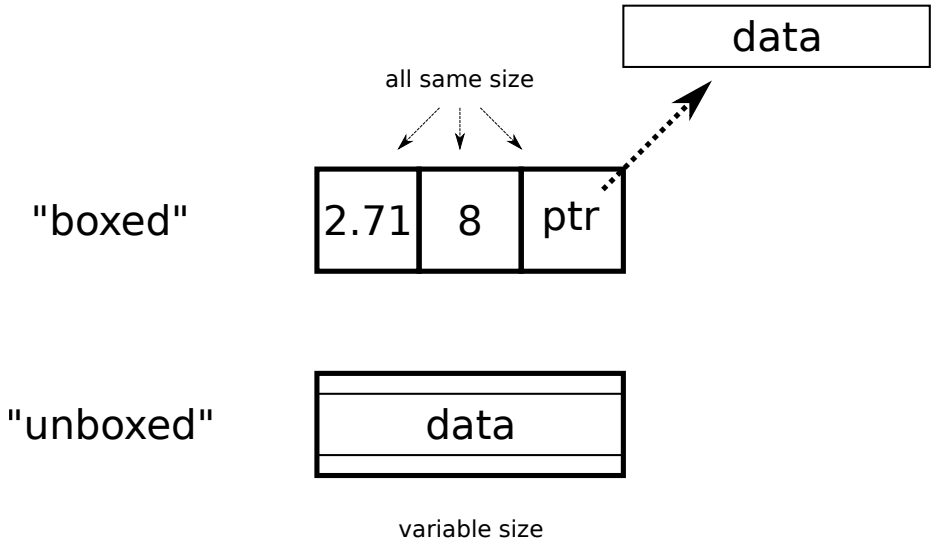

Figure 1.7: Boxed and unboxed values

```
typedef point2D = @{ x = double, y = double }
fun invert (p: point2D): point2D = @{ x = p.y, y = p.x }

// Roughly translates into C code of the following form:

struct point2D { double x, y; };
struct point2D invert (struct point2D p) {
  return (struct point2D) { .x = p.y, .y = p.x };
}
```

Listing 1.15: Flat record types

pointer-sized boxes. ATS supports programming with the same data representation as C, and the types that support these non-pointer-sized values are given the sorts `t@ype` and `viewt@ype` for regular and linear types respectively.

A common use-case for flat types is for storing records, or C structs. In ATS, you can denote a record type with the `@{ name = type }` syntax. For example, Listing 1.15 shows an example `point2D` flat type and how it might translate into C.

ATS also offers a positional record type known as a tuple, and a flat tuple is written like so: `@(x, y, z)`.

Flat types tend to be used a great deal when interfacing with C. For example, if

```
abst@ype point2D = $extype "struct point2D"
fun do_something (p: point2D): point2D = // ...
```

Listing 1.16: Referring to a C type

I had defined the `point2D` as a struct in C, and I wanted to refer to it abstractly in ATS, then I would write code like that shown in Listing 1.16. Although the internals of the type would be unavailable to ATS programs (without further elaboration), the ATS compiler would know that it was dealing with a type that did not have the same size as a pointer.

1.5.7 Templates

The convenience of having a boxed run-time representation is the ease of implementing polymorphism: when all values have the same size, then one piece of code can agnostically manipulate them without having to know further details. When dealing with unboxed representation, such as flat types, it is necessary to compile multiple versions of the code: for each size of value being handled.

For example, looking back at Listing 1.16, if we were to extend this `point2D` type into the third dimension, and create a `point3D` type, then that new type would have a larger size than the old one. Even though we cannot see the internal details of the type, the fact that it is larger means that the compiler must reserve more space when compiling functions that use it. For example, on some architectures, a value of type `struct point2D` could be stored using two machine registers, but a value of type `struct point3D` would require three machine registers. That could completely change the resulting assembly code, depending upon how many registers are needed and in what way they are used. Therefore, we need to create a separate version of our function that works on the larger type, as well as the one that works on the smaller type.

Doing all of that would be very inconvenient and repetitive when writing a function

```
fun{ty: t@ype} double (x: ty): @(ty, ty) = @(x, x)

typedef point2D = @{ x = double, y = double }

fun test (): void =
let
  val p1 = @{ x = 1.0, y = 2.0 }
  val p2 = double<point2D> (p1)
in
  // The type of p2 is @(point2D, point2D)
end
```

Listing 1.17: A simple template example

that is truly polymorphic. So ATS provides a feature known as "templates" to clean up and simplify the creation of such functions. A template is a function that is partially delayed in compilation: it is not fully compiled until it is actually applied in some use with a concrete type. The syntax is relatively simple and unobstrusive: when the type quantifier comes in between the keyword `fun` and the actual function name, it tells ATS that this is a template.

Listing 1.17 shows the use of a template named `double` that takes a single argument and returns a flat tuple containing 2 copies of the original argument. Because `double` is a template, it is not fully compiled until it is applied within the function `test` using the syntax `double<point2D>`. Then, a version of the template is created that supports the type `point2D`, and that version is compiled fully.

Templates allow us to work with ATS flat types in a generic manner, writing general code that can handle types of many different types. Also, in many cases, ATS can infer the template application, and does not require the explicit specification of the type using the `<...>` syntax. But it doesn't hurt. Oftentimes, the use of templates is almost indistinguishable from the use of functions.

1.5.8 ATS program structure

ATS code is commonly split up into several types of files:

- Files with the extension `.dats` are called "dynamics" and contain the general program code and implementations of functions.
- Files with the extension `.sats` are called "statics" and contain type definitions and function signatures.
- Files with the extension `.cats` are conventionally known as C glue files. The code within is written in C, and is expected to be connected to ATS definitions in a related `.sats` or `.dats` file.

Chapter 2

The Terrier Operating System

2.1 Platform support

Terrier began as a small kernel for the Texas Instruments OMAP3530 platform using an ARM Cortex A-8 single core processor. Basic memory management, scheduling, and I/O functionality was developed within this framework. The initial kernel project was given the name Puppy and released as an open source demonstration kernel, while the continued development of Terrier proceeded on a forked branch. Subsequently, I decided that the limitations of the OMAP3530 platform and Cortex A-8 processor were impeding interesting research, and I decided to port Terrier to the newer OMAP4460 platform using the multi-core Cortex A-9 processor. A widely available development platform is available with these specifications, known as the PandaBoard ES[1], and that is the platform on which Terrier is currently tested.

The PandaBoard ES offers the following features built-in to the board:

- Dual-core Cortex A-9 ARM processor
- 1 GB RAM
- Two high speed USB ports and one USB OTG port
- Integrated Fast Ethernet adapter, Wireless LAN and Bluetooth
- On-board RS232 serial port
- SD/MMC card reader
- Two HDMI display outputs

[1] `http://www.pandaboard.org/`

- 3-D graphics acceleration
- Audio jack input/output
- JTAG debugging
- Expansion connectors

All these and more make the PandaBoard a particularly potent platform for various kinds of control or data-processing applications: including but not limited to multimedia. It is also flexible enough to be used in several ways, which was important at the time since I did not know what applications would be used to test Terrier.

2.2 The boot process

2.2.1 Start-up

The kernel begins at the symbol `_reset` and begins to set up the various ARM modes, each of them receiving a dedicated stack in order to have clean transitions later when needed. However, if Terrier is compiled with virtual memory support, then those stack addresses must be virtual, so in that case the very first code executed is a piece of C code in the file `init/stub.c`. This code sets up a very simple page table with two mappings: first, an identity mapping so that execution can proceed immediately after enabling virtual memory; and secondly, it establishes the main kernel mappings to high memory, so that the start-up code can jump directly into the early initialization function written in C.

2.2.2 Early initialization

The early initialization function found in `init/init.c` is written in C because it is not particularly interesting. It is simply a sequence of calls to initialization functions in other subsystems. A few pieces of hardware are initialized right away because they

are useful to the early boot process: the performance monitoring unit, the serial port, and the memory management infrastructure.

2.2.3 Physical and virtual memory

The physical memory initialization process found in `mem/physical.cats` requires probing a few platform-specific registers to learn the size of installed memory and a few of its properties. Then, the bitmaps that track the status of free physical frames of memory are initialized. The frames are all marked as available, except for the ones needed by the physical memory allocator and the ones used by the kernel. These frames are determined with the help of linker symbols and a few calculations to translate the numbers into the actual physical addresses.

The virtual memory subsystem in `mem/virtual.c` assumes that virtual memory has already been enabled during the stub procedure. Therefore its task is largely to initialize and configure the second-level pagetable that the kernel uses to manage kernel-space memory. It also takes advantage of an ARM processor feature that allows you to use two separate page directories: one of the directories covers the memory space below the 1 GB mark, and the second one covers the space above 1 GB. With the kernel conveniently mapped above 1 GB, this allows us to deal with user-space and kernel-space virtual mappings in entirely separate page directories. Finally, the translation lookaside buffer is flushed to ensure there are no stale mappings.

2.2.4 Multiprocessor support

Initialization

The Cortex-A9 supports multiple processors, as does Terrier. The operating system begins execution upon a single bootstrap processor and must kickstart the auxiliary processors. In `omap/smp.c`, function `smp_init`, the first thing is to find the number of other cores and prepare shared variables for the initialization process. It is possible

to boot all the auxiliary processors at the same time, but I have chosen to do it one at a time for simplicity. On the ARM processor, it is necessary to bring up auxiliary CPUs in a specified and controlled sequence in order to get them running the right pieces of code and to enjoy features such as cache coherency. Therefore, for each processor, the bootstrap processor follows a particular sequence of steps.

1. The desired auxiliary CPU index is stored in the shared variable `curboot` and shared variable `stage` is initialized to 0.

2. Each CPU needs to have its own set of stacks for managing processor modes, so the bootstrap CPU allocates a set of pages and stores them into the shared variable `newstack`.

3. The register `AUX_CORE_BOOT` allows us to specify a starting address for the auxiliary CPU. That is configured to be `smp_aux_entry_point`.

4. `data_sync_barrier` ensures that everything is written to memory before proceeding.

5. The default ARM processor state is called "wait-for-event" (`WFE`) and that can be tripped by having any processor invoke the ARM instruction "set-event" (`SEV`). This causes all waiting processors to resume execution.

6. Then begins a two-sided sequence controlled by the `stage` variable:

Stage 0 All processors execute `smp_aux_entry_point` but only the designated one successfully passes the `curboot` check. The others are set back to sleep. The freshly woken processor then goes and sets up its processor mode stacks, switches back to supervisor mode, and then invokes its first real function call to `smp_aux_cpu_init`, where it indicates that Stage 0 is complete and it waits for the bootstrap processor.

Stage 1 The bootstrap processor enables the Snoop Control Unit, which is responsible for cache coherency.

Stage 2 The auxiliary processor obtains its affinity number and checks a few variables for sanity.

Stage 3 The bootstrap processor enables SMP.

Stage 4 The auxiliary processor enables SMP.

Stage 5 The auxiliary processor cleans and initializes its caches, while the bootstrap processor goes ahead and initializes all the other processors and also its own caches.

Stage 6 All caches and processors are initialized.

7. The auxiliary processor then goes ahead and initializes its own processor-specific needs in parallel to everything else, similar to the bootstrap init process, but only focusing on itself: performance monitoring, interrupt handling, per-CPU variables, and the dual page directory feature.

8. At this point the auxiliary processor is ready for operation and it waits on a semaphore for the scheduler to be enabled. When that happens, the auxiliary processor will begin in a designated "idle process" before it is assigned its first real task.

2.3 Hardware support

2.3.1 Memory

Physical memory

The physical memory manager is found in `mem/physical.{dats,sats,cats}`. A set of bitmaps are maintained to track the free and used frames of memory: each frame, or physical page, being four kilobytes in size. Physical memory is a resource managed

by the kernel. In ATS, it is kept fairly simple, a simple indexed wrapper around the raw address:

```
abst@ype physaddr_t (p: addr) = int
typedef physaddr = [p: addr] physaddr_t p
```

When designing the types I had a choice to make: should physical addresses be treated as resources to be managed with linear types? I chose not to do so because most uses of physical addresses in the kernel have indefinite extent, and it was not a problem that needed the oversight of ATS.

The primary interface to the manager is via the `physical_alloc_pages` function, shown here with an ATS prototype:

```
fun physical_alloc_pages (
  n: int,
  align: int,
  addr: & physaddr? >> physaddr_t p
): #[s: int] #[p: addr | s == OK <==> p > null] status s
```

This function attempts to allocate `n` frames of memory aligned by `align` frames. If successful, it returns a status of `OK` (0) and stores the physical address into the output parameter named `addr`. The type of the output parameter `physaddr?` has a question mark modifier initially, meaning that it is an uninitialized location, but it will be initialized upon return. The return type asserts that if the status is not `OK` then the physical address stored will be the `null` address. This forces callers of this function to perform a safety check to ensure that it actually did succeed.

There is also a `physical_free_page` function that unmarks the page in the bitmap. It is straightforward and returns no value. Further discussion of the implementation of the physical memory manager may be found in Section 4.2.

Virtual memory

The virtual memory manager is in `mem/virtual.c` and is only available when the kernel is compiled with virtual memory support. The original code was written in C

with extensive run-time safety checks, and since safety nor correctness never became an issue, conversion to ATS was mooted. The virtual memory interface is considerably more flexible than the physical memory interface. It is based on the underlying ARMv7 Virtual Memory System Architecture and there are two levels currentl supported: 1 MB pages and 4 kB pages.

The interface relies upon two descriptive data structures found in `mem/physical.h`: `struct pagetable` and `struct region`. In order to introduce a new pagetable into the system the following structure must be filled out:

```
typedef struct pagetable {
  void *vstart; /* starting virtual address managed by PT */
  physaddr ptpaddr; /* address of pagetable in physical memory */
  u32 *ptvaddr; /* address of pagetable in virtual memory */
  struct pagetable *parent_pt; /* parent pagetable or self if MASTER */
  u16 type; /* one of PT_MASTER or PT_COARSE */
  u16 domain; /* domain of section entries if MASTER */
} pagetable_t;
```

Once a pagetable has been initialized you can associate a virtual "region" with it by filling out this data structure:

```
typedef struct {
  physaddr pstart; /* starting physical address of region */
  void *vstart; /* starting virtual address of region */
  pagetable_t *pt; /* pagetable managing this region */
  u32 page_count; /* number of pages in this region */
  u16 page_size_log2; /* size of pages in this region (log2) */
  u8 cache_buf:4; /* cache/buffer attributes */
  u8 shared_ng:4; /* shared/not-global attributes */
  u8 access; /* access permission attributes */
} region_t;
```

Mapping a region and activating the pagetable will put the region into effect. The primary interfaces are the following functions:

```
status vmm_init_pagetable(pagetable_t *pt);
status vmm_activate_pagetable(pagetable_t *pt);
status vmm_map_region(region_t *r);
status vmm_map_region_find_vstart(region_t *r);
status vmm_get_phys_addr(void *vaddr, physaddr *paddr);
```

Other than "get physical address", the functions operate on the two data structures shown above. All of them return a status, as described in `status.h`. Pagetable initialization means checking the structure for consistency and then clearing out the entries. Activating a pagetable means either configuring the page directory (`TTB0`) for `MASTER` level tables, or writing the appropriate entry for `COARSE` level tables. Mapping a region will cause the pagetable to be modified so that the region is expressed via pagetable entries. And the second mapping function, suffixed with `_find_vstart`, is for when you do not care what virtual address is used, you just want the manager to find you a set of unused virtual addresses that fit the region.

2.3.2 Timers

The OMAP4460 platform offers a selection of 11 general-purpose timers, and a private per-CPU timer known as `PVTTIMER`. The driver in `omap/timer.c` wraps a simple interface around some basic memory-mapped register manipulation for general timer purposes, and `omap/timer.h` adds a scheduler-oriented wrapper around the `PVTTIMER` functionality. General-purpose timer IRQ numbers are mapped from 37 through 47, while the `PVTTIMER` IRQ is number 29. ARM also provides a global counter that always increases at a known constant rate of 32 kHz, until it overflows and resets over again. A simple spin-wait function that handles overflow, named `timer_32k_delay`, is made available for primitive code.

Use of `PVTTIMER` requires measurement of its rate, which in Terrier is conducted during timer initialization. The `PVTTIMER` rate is measured using both the builtin 32 kHz clock, `timer_32k_value`, as well as the ARM cycle counter, `arm_read_cycle_counter` (for informational purposes). This computes a prescaler value based on the measurements. That prescaler value is later used when setting the timer value for scheduling purposes.

Abstract private timer interface for schedulers:

```
void pvttimer_set(u32 count);
u32 pvttimer_get(void);
void pvttimer_enable_interrupt(void);
void pvttimer_disable_interrupt(void);
void pvttimer_ack_interrupt(void);
u32 pvttimer_is_triggered(void);
void pvttimer_start(void);
void pvttimer_stop(void);
void pvttimer_set_handler(void (*handler)(u32));
```

which are all quite simply implemented as one-line inline functions in C, for performance.

2.3.3 Serial port

Serial port support is provided by the NS16650 UART on the OMAP4460 platform. This particular UART is identified as UART3 in the technical reference manual. The memory mapped registers are manipulated by code found in `init/early_uart3.c`. As the name implies, this code is intended largely for debugging purposes, because the serial port is the major input/output device available to the system from a very early stage. The primitive functions available are `putc_uart3`, `putx_uart3`, and `print_uart3`, which respectively print a character, a hexadecimal number, and a string through the serial port.

Debugging output

The primitive serial port functions are awkward to use. Instead, debugging output is directed through the debugging subsystem found in `debug/log.c` and `debug/log.h`:

```
void debuglog(const char *src, int lvl, const char *fmt, ...)
void debuglog_no_prefix(const char *src, int lvl, const char *fmt, ...)
void debuglog_dump(const char *src, int lvl, u32* start, u32* end)
void debuglog_regs(const char *src, int lvl, u32 regs[16])
```

These functions provide convenient, printf-like functionality available for output through the serial port, controllable by debugging levels, and tagged by a source

module. Furthermore, convenience macros `DLOG`, `DLOG_NO_PREFIX`, ..., obviate the first parameter when the `MODULE` preprocessor variable is defined. Most files define a `MODULE` at the top, making `DLOG` the most typically used debugging output interface.

2.3.4 USB

The USB subsystem has been delegated out into program space. The `ehci` program implements a portion of the Enhanced Host Controller Interface that is used by the OMAP4460 platform high-speed USB host controller. This code is found in `progs/ehci/ehci.{dats,sats,cats}`. Initialization of the USB subsystem requires the manipulation of a variety of different, esoteric hardware modules on the OMAP4460 platform, primarily to enable power systems and clocks. Once that is accomplished, the memory-mapped registers of the standard EHCI design may be accessed and manipulated according to the standard EHCI specification.

The `ehci` module defines several data structures to help manage USB operation. The `struct usb_root_port_t` describes and helps provide access to the root ports of the host controller – up to 3 in this case. The `struct usb_hub_port_t` accomplishes the same for any discovered hub ports. USB devices, in general, are stored in a directory composed of `struct usb_device_t` data structures. One of the interesting features of this EHCI driver is that these fundamental `usb_device_t` structures are stored in an exported directory that is accessible via IPC mechanisms. This is an inside-out approach to device driver interfaces that ostensibly exposes the internals of the driver for manipulation by other programs. The safety of this mechanism relies upon the static checking performed by the IPC protocol code, that code being written in ATS.

Back in the driver itself, upon initialization, it begins to identify attached devices using the `ehci_enumerate` function and calls `ehci_setup_new_device` whenever it finds one. When hubs are found they are recursively enumerated. The remainder of the file `ehci.cats` is primarily concerned with defining the standard USB data structures

"Queue Head" (QH) and "Transfer Descriptor" (TD) as well as the basic configuration functions required to perform USB control and bulk data transfers. However, it is not anticipated that most programs will use these features directly, since they are part of the `ehci` module internally. Instead, programs are expected to define their own, or more likely, use a set of library-defined functions, that perform the necessary data structure manipulation required to send and receive data over USB.

The EHCI specification defines a circularly-linked set of QHs as being the place for the host controller hardware to find bulk and control transfers to execute. A memory-mapped register named `EHCI_ASYNCLISTADDR` points to one of the QHs, and the hardware expects to be able to cycle through the circularly-linked list of QHs from there, while it is running. Therefore, it is important to maintain the consistency of the QH structures. Typically, each QH corresponds to a USB device, because it is configured with a particular USB address. A chain of TDs is constructed that describes a particular transaction, and then that transaction is executed by setting a field in the QH as a pointer to the first TD in the chain.

Individual programs that manage their own USB device can independently construct and attach a TD chain to the corresponding QH. Therefore, the IPC interface that manages the USB devices merely has to permit each program to manipulate the TD pointer in its designated QH, while preventing access to the TD pointers in the other QHs, and while protecting the overall circularly-linked list of QHs. This is a task that ATS can handle at the type level, as described in Chapter 4.1.

2.4 Scheduling

2.4.1 The process model

Program entry handlers

One of the distinguishing features of Terrier from other operating systems is the way it handles processes. A principle of Terrier from the beginning has been to remove

some uses of hardware barriers or run-time checking in favor of putting more responsibility on the programmer. The goal is to allow programmers the opportunity to achieve better responsiveness and performance, as well as to explore models of programming that are not traditionally supported at the operating system level. Instead of providing an abstraction that might have to be worked around by the application programmer, I would like to provide a model that is more closely tied to the way hardware actually behaves. So instead of pretending that programs execute seamlessly, I give them an opportunity to respond to these kinds of arbitrarily-timed control transfers by employing a designated handler: a piece of code that I call the "entry handler" because I have decided that it makes the most sense for the beginning of the program to also serve as the handler. Whenever a program is interrupted, it is not eventually resumed, but rather, control is transfered to the entry stub. The previous context – the one that was interrupted – is made available for use by the entry handler, but it is not required to be used. Instead, applications are given the opportunity to make their own decision about whether or not they would like to resume, or do something else.

The result is much more flexible: an application program could implement its own preemptible thread library without kernel assistance, using this mechanism. The application is treated as a full partner, together with the kernel, in making decisions about the continuation of the program after interruption. The continuation is provided in the form of a data structure that contains the program context at the point of interruption. A program may instead choose to save that context and load a different one, in order to create the same effect as a kernel-based thread library.

One immediate concern that may be raised with the entry handler mechanism is: What happens if the entry handler itself is interrupted? Since entry stubs are not part of the kernel, but rather part of the application, they do not have the ability to shut off or mask interrupts. However, shutting off or masking interrupts is also counter to

the goal of providing a responsive real-time system. Again, I have sought a software solution in place of a hardware solution. I have designated the entry handler to be a special section of the program for which interrupts do not save context. Therefore, any existing context that is saved by the kernel will not be overwritten when the entry handler is interrupted. However, as a result of this condition, entry handlers must be programmed in such a manner that they are both reentrant and restartable. In other words: the entry handler could potentially be interrupted and restarted at any moment, multiple times. Therefore, the code that runs as part of the entry handler must be able to tolerate these conditions and still perform its duties.

It is also best if the entry handler has as few instructions as possible, in order to work most efficiently. Therefore I elected not to write entry handlers in ATS, but rather, to hand-code them in ARM assembly language. Although it is possible to use ATS to generate assembly code via embedded C and inline assembly, I felt that it would be more useful and practical to apply model-checking techniques for the purpose of verification in this instance. The properties that I wish to verify in this instance are relatively limited and the typical entry handler is written with simple techniques resembling a finite state automata. Therefore, model-checking is a good fit. I describe the use of model-checking to verify certain properties of the entry handler in Chapter 5.2.

Prior to any verification effort, I will show a simple example of an entry handler. In Listing 2.1 is an example of the most basic possible entry handler, the one that is used by default in Terrier programs.

Some explanation is in order: `_start` is the traditional name for the symbol that indicates the beginning of a program. `.section .entry` tells the linker to put this piece of code in a specially marked section that the kernel will understand to be the entry stub. Then begins the assembly language code: `CMP r13, #0` tests the value of general-purpose register `r13` against the literal constant `0`, and sets the Zero Flag if

```
        .globl _start
        .section .entry
_start:
        CMP r13, #0 /* r13==0 iff initial entry */
        LDREQ r13, =_stack
        LDREQ r15, =main

        LDR r0, [r13]
        MSR cpsr_sf, r0 /* restore cpsr (status, flags) */
        ADD r13, r13, #8
        LDMIA r13, {r0-r15} /* load context, branch to it */

        .section .bss
        .space 1024
_stack:
```

Listing 2.1: Default entry handler

they are equal. The reason for this test is that the entry handler protocol is defined in the following manner:

- If `r13` is 0 upon entry then this is the initial invocation of the program.
- If `r13` is not 0 then it must be equal to a valid pointer to a space, accessible by the program, containing the previously interrupted context of the program.

The `LDREQ` instruction is actually the load instruction `LDR` annotated with a special suffix `EQ` that will only allow the instruction to execute on the condition that the Zero Flag is set. This is a feature of the ARM instruction set called "conditional execution" that allows programmers to compactly branch program flow without creating additional labels and branch instructions. Therefore, if the Zero Flag is set due to `r13` being found to be equal to 0, then the following two instructions will first assign `r13`, the conventional stack pointer, to be equal to top of the stack, and will then branch to the `main` function by changing the value of the program counter, `r15`, directly.

In the case that the Zero Flag is not set because `r13` is not equal to 0, then execution will fall through to the following instruction `LDR r0, [r13]`. If `r13` is not 0 then it is presumed to be a valid pointer to a context. The first word in a context is

the saved Program Status Register. This must be restored using the `MSR` instruction as shown in the listing. Then the pointer contained in `r13` is advanced by 8 bytes so that it is pointing at the saved values of `r0` through `r15`. At this point, we can take advantage of another feature of the ARM architecture named "Load Multiple" or `LDM`. The suffix `IA` means "Increment, After each load" and the `LDMIA` instruction takes a pointer and loads a sequence of values into the specified registers, following the ordering specified by the suffix. In this particular instance, it loads all 16 registers simultaneously, from the memory address pointed to by `r13`. That implies that the value of `r15` is also loaded, and the ARM architecture treats a load into `r15` as an immediate control transfer to the location specified by whatever value is loaded. Therefore, the entire register context of the program can be loaded and resumed in one single instruction. This feature of the ARM architecture was found to be particularly useful during the design of Terrier.

So, the default entry handler does not take advantage of the power or flexibility offered by this arrangement at all. Instead, it simply behaves like a more traditional operating system, and resumes the program from wherever it left off. The entry handler is simple enough that it should be possible to see from cursory examination that it is both reentrant and restartable under the preconditions described earlier. If, at any point, it is interrupted, it simply begins the process of loading the context again, next time it gets to run.

2.4.2 Program file format

Support for the ELF

Terrier has built-in support for loading and parsing files in the standard Executable and Linkable Format. Executables are expected to be linked in accordance with the rules found in the linker script `ldscripts/program.ld`. The basic memory layout of a program is shown in Table 2.1.

```
.text
        *(.entry)
                        _end_entry
        *(.text)
                        _etext
.rodata
        *(.rodata)
.data
        *(.data)
.bss
        *(.bss)
.device ALIGN(0x1000)
        *(.device)
```

Table 2.1: Terrier executable program memory layout

`_scheduler_capacity`	The capacity parameter p_C in 32kHz ticks.
`_scheduler_period`	The period parameter p_T in 32kHz ticks.
`_scheduler_affinity`	The CPU affinity.
`_kernel_saved_context`	Address of memory block used by kernel to save the program's context.
`_mappings`	`NULL`-terminated array of mapping specs.
`_ipcmappings`	`NULL`-terminated array of IPC mapping specs.
`_end_entry`	End of special entry handler code section as described by Section 2.4.1.

Table 2.2: Symbols that Terrier interprets specially

Terrier also takes advantage of ELF in order to statically specify parameters of each program, and implement some features of the OS. Those symbols are shown in Table 2.2.

2.4.3 Scheduling parameters

The rate-monotonic scheduling algorithm is described in Section 2.4.6, which further details how `_scheduler_capacity` and `_scheduler_period` are used. The value of the `_scheduler_affinity` global variable is used to choose the processor on which the program is executed.

```
typedef struct {
  physaddr pstart;
  void *vstart;
  u32 page_count;
  u16 page_size_log2;
#define R_B 1
#define R_C 2
  u8 cache_buf:4;
#define R_S 1
#define R_NG 2
  u8 shared_ng:4;
#define R_NA 0
#define R_PM 1
#define R_RO 2
#define R_RW 3
  u8 access;
  char *desc;
} mapping_t;
```

Listing 2.2: The `mapping_t` struct

2.4.4 Mappings

The C struct that describes a mapping is shown in Listing 2.2. Mappings allow a program to request that the Terrier memory system provide access to a specified window of physical memory. In fact, since Terrier can run with or without virtual memory enabled, the mapping system must handle both cases. When virtual memory is enabled, during load-time, Terrier will try to create a virtual mapping of the window, and store the resulting virtual address in the `vstart` field. If virtual memory is not enabled, then Terrier will simply copy the value of `pstart` into `vstart`.

The mapping struct allows the program to request certain properties of the virtual memory mapping. There is the number of pages requested, `page_count`, and the size of pages in $\log_2$ terms, `page_size_log2`. The program can also request that the memory be cached or buffered using `cache_buf`, and that it be shared or "not global" using `shared_ng`. Finally, `access` can be restricted by mode, and a `desc` is useful for interpreting kernel messages. An example set of mappings is shown in Listing 2.3. The last mapping should always be a `NULL` mapping.

```
mapping_t _mappings[] = {
  { 0x48050000, NULL, 10, 12, 0, 0, R_RW, "needed for GPIO" },
  { 0x4A064000, NULL, 1, 12, 0, 0, R_RW, "EHCI base" },
  { 0x4A009000, NULL, 1, 12, 0, 0, R_RW, "CM base" },
  { 0x4A062000, NULL, 1, 12, 0, 0, R_RW, "USBTLL base" },
  { 0x4A064000, NULL, 1, 12, 0, 0, R_RW, "HSUSBHOST base" },
  { 0x4A310000, NULL, 2, 12, 0, 0, R_RW, "needed for GPIO" },
  { 0x4A30A000, NULL, 1, 12, 0, 0, R_RW, "SCRM base" },

  { 0 }
};
```

Listing 2.3: Example of mappings

2.4.5 Interprocess communication mappings

The IPC mapping feature, with struct definition in Listing 2.4, helps programs establish communication channels. These mappings are then made available to application programs through an ATS interface discussed in Section 3.2. The mappings are specified at a higher level than the regular mapping feature, which is primarily intended to help programmers write device drivers. With the IPC mapping feature, Terrier creates a database of programs and potential channels, and uses it to match them up as IPC partners. So for example, if a program defines an IPC mapping that is "seeking" another mapping by the name of "uart", then Terrier will find a program that is "offering" a mapping by the name of "uart" and pair them up, if they both agree to follow the same protocol. A successful pairing means that a piece of memory is allocated, of `pages` length, and then that memory is mapped into both program's spaces. Each program is informed about the mapping through the `address` field in the struct. If virtual memory is enabled, then it is quite possible that the integer value of `address` will be different in each program – but if virtual memory is disabled then both will share the same address.

The `type`, `name`, and `proto` fields allow this search process to take place as described above. The `flags` field adds some properties of the channel, for example if the program

```
typedef struct {
#define IPC_SEEK 1
#define IPC_OFFER 2
#define IPC_MULTIOFFER 3
  u32 type;
  char *name;
#define IPC_READ 1
#define IPC_WRITE 2
#define IPC_ALWAYSALLOC 4
#define IPC_DEVICEMEM 8
  u32 flags;
  char *proto;
  u32 pages;
  void *address;
} ipcmapping_t;
```

Listing 2.4: The `ipcmapping_t` struct

intends to use it as a "read" or a "write" style channel. The `IPC_ALWAYSALLOC` flag tells Terrier to always allocate a piece of memory for this mapping, even if it cannot find a partner. That's useful for programs that store crucial data structures in the IPC shared memory region, even if they do not get paired up with another program. Finally, the `IPC_DEVICEMEM` flag indicates that the IPC shared memory region is not only being used by another program, but it could also be accessed by a hardware device.

An example of IPC mappings is shown in Listing 2.5. This example seeks out the "uart" channel using the "fourslot2w" protocol, and it offers two channels for other seekers: "ehci_info" using "fixedslot" and "ehci_usbdevices" using a custom protocol named "usbdevices" that is flagged for `IPC_ALWAYSALLOC` and `IPC_DEVICEMEM`. That's because this last shared memory region will be used for storing data structures used by hardware devices.

2.4.6 Rate-monotonic scheduling (RMS)

Consider a system where you have a set of tasks that are statically assigned properties such as *capacity*, C_i, and *period*, T_i. Then, a *rate-monotonic scheduling algorithm* is

```
ipcmapping_t _ipcmappings[] = {
  { IPC_SEEK, "uart", IPC_WRITE, "fourslot2w", 1, NULL },
  { IPC_OFFER, "ehci_info", IPC_WRITE, "fixedslot", 1, NULL },
  { IPC_OFFER, "ehci_usbdevices",
    (IPC_READ | IPC_WRITE | IPC_ALWAYSALLOC | IPC_DEVICEMEM),
    "usbdevices", 1, NULL },
  { 0 }
};
```

Listing 2.5: Example of IPC mappings

one that treats shorter period tasks as having higher priority than longer period tasks. The period of a task means how much time there is between activations, and the capacity is the amount of time that the task is allotted to run within that period. Therefore, the ratio $U_i = C_i/T_i$ is often referred to as the processor *utilization* of the task i, and $0 \leq U_i \leq 1$. RMS was introduced and analyzed by Liu and Layland (1973), and it has remained an important algorithm for static priority real-time scheduling ever since. They proved that for a set of n tasks with n different periods, it is possible to show that there is always a feasible schedule if the sum of utilizations falls below a certain bound. That is,

$$\sum_{i=1}^{n} U_i \leq n \left(\sqrt[n]{2} - 1 \right)$$

and that this formula converges at $\ln 2 \approx 0.693$, meaning that any such set of tasks with $\sum U \leq 0.693$ can be guaranteed a feasible schedule.

The default scheduler of Terrier is based on RMS, although it is parameterized and can be swapped out at compile-time for the purpose of experiments. A common problem with RMS is that it is easily subject to *priority inversion* when multiple tasks of different priorities participate in locking protocols. Terrier's main priority inversion avoidance strategy is to rely on lock-free algorithms for communication between tasks, as described in Section 3.3. The RMS scheduler is implemented in `sched/rms_sched.{dats,sats,cats}`. The implementation of a task is referred to as a process within the context of Terrier. The primary function interface includes

`sched_wakeup`, which gives a process a budget equal to its full capacity, and `schedule`, which updates all the state and chooses the next process to run.

Each process p is given the following properties:

- p_C, p_T are the capacity and period of p.
- p_b is the budget, or remaining time left to run, for p.
- p_r is a time value indicating when the budget is to be refilled.

The pseudo-code for the RMS scheduler is as follows:

1. Look up current running process, named p.
2. Find out how much time t_{span} has passed since previous context switch.
3. Subtract t_{span} from p_b, setting the budget to zero if it is below a negligible amount.
4. For each process q in the system, check the replenishment time:
   (a) If $t_{now} \leq q_r$ then increment q_r by q_T and set $q_b \leftarrow q_C$.
5. Select the process with positive budget and smallest period, call it p_{next}.
6. Select a time value t_{val} based on all the replenishment times, or the expiration of the budget of p_{next}, whichever is smallest.
7. Mark p_{next} as context switch target, which will occur upon return to user-space.
8. Set the processor timer to wake-up at t_{val}.

For the purpose of testing, each process selects a CPU affinity ahead of time, and so the schedulers operating on separate processors do not interfere with each other's data structures.

2.4.7 Interrupt handling

Section 2.4.1 described the mechanism with which Terrier begins and restarts interrupted processes. Upon restarting, the context that was interrupted is made available through a pointer stored in `r13`. In addition, a table of interrupt statuses is also made available through a pointer stored in `r12`. Each byte in this table corresponds to an interrupt. The first bit is toggled on when the interrupt is active. The remaining bits are reserved for future use. An address for the interrupt status table is also available through invocation of instruction `SWI #1`, which will store the address of the table in the saved `r0` register. An entry handler can be designed to check the bit in the interrupt status table and take action accordingly. An example of such an entry handler is discussed in Section 5.2.

Chapter 3

Interprocess Communication Mechanisms

3.1 Introduction

Terrier provides individual programs with a great deal of power and flexibility to work within themeselves, but it also provides a set of mechanisms for communicating between programs. The Terrier philosophy favors loosely-coupled interfaces that avoid synchronization and its associated problems. Actual communication itself should be able to avoid invoking kernel functionality, instead relying on a set of libraries that implement various protocols in ATS. The kernel only provides the channels that will be used, and they are specified statically by mappings as described in Section 2.4.5. The interface to the raw memory is described in the following section, and the various library mechanisms that use the memory are described in the remainder of this chapter.

3.2 Interprocess communication memory interface

Section 2.4.5 described the specification mechanism for interprocess communication (IPC) mappings, and this section describes the programming interface in ATS made available for programmers to use those mappings. All of the communication mechanisms described in this chapter make use of this IPC mapping interface for obtaining access to the IPC memory.

Listing 3.1 shows the interface, which has been kept relatively brief. The view `ipcmem_v` acts as the proof that a certain pointer over a certain range of pages is entitled to be a handle for IPC memory. Since IPC memory is shared between processes and processors, it has been specially set up for that purpose, and the pointer can only be

```
absview ipcmem_v (l: addr, pages: int)

fun ipcmem_get_view (
  id: int, pages: & int? >> int pages
): #[pages:nat] [l:agez] (option_v (ipcmem_v (l, pages), l > null) | ptr l)

prfun ipcmem_put_view {l:addr} {pages:int} (_: ipcmem_v (l, pages)): void
```

Listing 3.1: IPC memory interface

obtained through a call to `ipcmem_get_view`. The first parameter to this function is the "identifier" for the IPC memory. I have chosen to keep this simple, using only an index number into the mapping array, but in the future it might make sense to devise a naming scheme.

The `ipcmem_get_view` function returns a pointer, and it also modifies the second parameter by-reference to store the number of pages available in the IPC memory. But the pointer is not available to be used as IPC memory until it is checked for the `null` value. That is the meaning of `option_v (ipcmem_v (l, pages), l > null)`: the view can only be unpacked from the `option_v` if we can suitably show that `l > null`. This is a means to force programmers to check for null-pointers before proceeding to use the value in more dangerous ways. Because all of this proof work occurs at the type-level, it is all erased by the ATS compiler, and therefore poses no additional run-time overhead apart from a normal null-pointer check.

Finally, the counterpart to `ipcmem_get_view` is `ipcmem_put_view`, which has been left as a proof-function because it currently has no real effect under the implemented protocol, and can be safely erased entirely during compile-time.

3.3 Asynchronous communication mechanisms

An asynchronous communication mechanism (ACM) is a means for two programs to exchange a piece of data without relying on explicit synchronization. By avoiding the use of locks or semaphores, programs can interact safely without fear of deadlock,

priority inversion, or unpredictable blocking delay being caused by OS-level code. Many such mechanisms have been described in past work. Simpson (2003) created a taxonomy of such protocols, out of which I am most interested in the *pool*-style protocols. In his telling, these pool-style protocols are designed to exchange a piece of *reference data* between some number of writers and some number of readers. The writers may destructively overwrite the reference data at any time. The readers may non-destructively read a coherent copy of the reference data at any time. The details of the protocol ensure that the writers and the readers do not interfere with or confuse each other. Pool-style ACMs typically strive to achieve the following properties:

Coherency Reading and writing operations should not trample on each other. A writer should always be able to write data cleanly without interference. And a reader should always be able to obtain a complete piece of data exactly in the same form that an earlier writer placed it.

Freshness Readers should be able to access the most recent, coherent piece of data as soon as possible after the writer has finished putting it into memory. Readers should never see a past version of the data once a fresher version has been made available.

Concurrency Reading and writing operations should be able to run as concurrently as possible, without interference from each other.

3.4 The four-slot mechanism and generalizations

3.4.1 Introduction

The "four-slot mechanism" (Simpson, 1990) is an ACM designed to allow one-way, memory-less communication between a writer program and a reader program with-

out the use of locks. This mechanism was later generalized to multiple readers and multiple writers (Simpson, 1997a).

The following sections use the formulation of the mechanism that was developed in the second paper, a form which varies significantly from the original, but is more easily generalized. First is shown a proof of coherency for the single-reader version. Then it is expanded and shown that each of the steps has a "linearization point" (Herlihy and Wing, 1990). Finally, a similar proof is elaborated for the multiple-reader version of the mechanism. In the future, a multiple-writer version will be added.

Past work by Simpson (1997b) has focused on model-checking techniques to show correctness of the mechanism. This section contributes an alternative formulation of the coherency proof, given in the style that will be used for other algorithms encoded into ATS and provided with Terrier. It is a style that blends lemmas worked out by hand on paper along with reasoning encoded into ATS types and checked by machine.

3.4.2 Definitions

Definition 1 (Potentially conflicting steps)**.** R1 *and* W3 *are examples of operations that take time to read or write the data array, and therefore, could conflict with each other if they are both operating on the same element of the array during overlapping periods of time.*

Definition 2 (The "precede" relation)**.** *When operation A is said to precede operation B, it means that operation A occurs prior to the execution of operation B, and the full effect of operation A is visible and available to operation B.*

Definition 3 (Interacting operations)**.** *If two operations, A and B, act upon the same memory address then they are said to be interacting. If those operations are linearizable then the ordering is strict: either A precedes B or B precedes A.*

Definition 4 (Associated operations)**.** *The operations which set up the state variables in preparation for the subsequent read or write are considered to be "associated" with that read or write. For example, instances of* W2 *and* W3 *are associated with the subsequent instance of* W1*. And instances of* R1 *and* R2 *are associated with the subsequent* R3*.*

3.4.3 Single-reader mechanism definitions

- r, w are shared bit-arrays, $|r| = |w| = 2$.
- ip, op are shared bit variables
- $d[\cdot, \cdot]$ is the four slot array indexed by two dimensions

Property 1. *Steps* W2, W3, R1, *and* R2 *are all linearizable.*

Reader pseudo-code

R1 $r \leftarrow w$

R2 $op \leftarrow \neg ip$

R3 read data from $d[op, r[op]]$

Writer pseudo-code

W1 write data into $d[ip, w[ip]]$

W2 $ip \leftarrow \neg ip$

W3 $w[ip] \leftarrow \neg r[ip]$

3.4.4 Single-reader proofs

Reader

The following lemmas apply to the reader code and are applicable until the reader completes R3.

Lemma 1. *Given an instance of* W3*, if it does not occur in the time between the beginning of* R1 *and the completion of* R3*, then the associated* W1 *is non-conflicting because* $op \neq ip$ *or it does not overlap with a conflicting* R3*.*

Proof. Suppose there is a W1 which potentially conflicts with R3, but the associated W3 operation occurred preceding the R1, as shown in Figure 3.1. Then by transitivity, it must be the case that the associated W2 preceded R2, and therefore it is safe to say that $op \neq ip$ because R2 sets $op \leftarrow \neg ip$. □

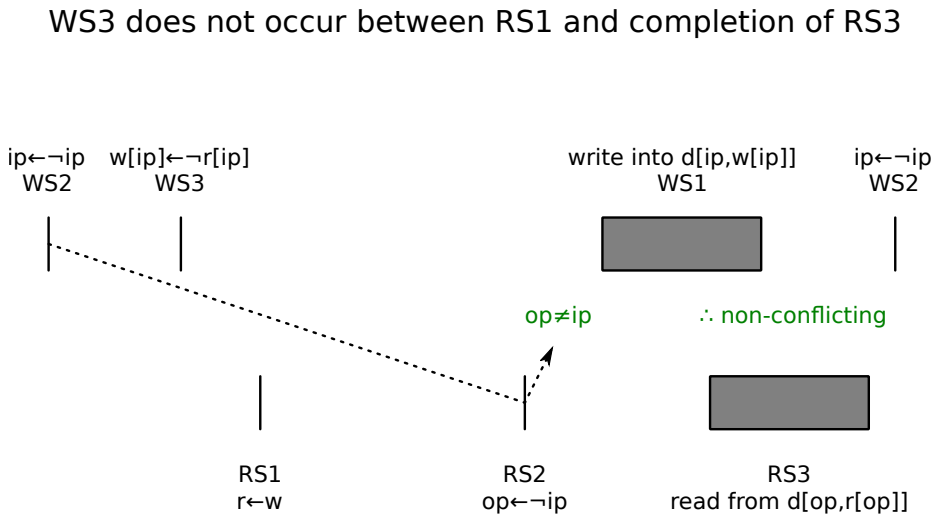

Figure 3.1: W3 does not occur between R1 and completion of R3

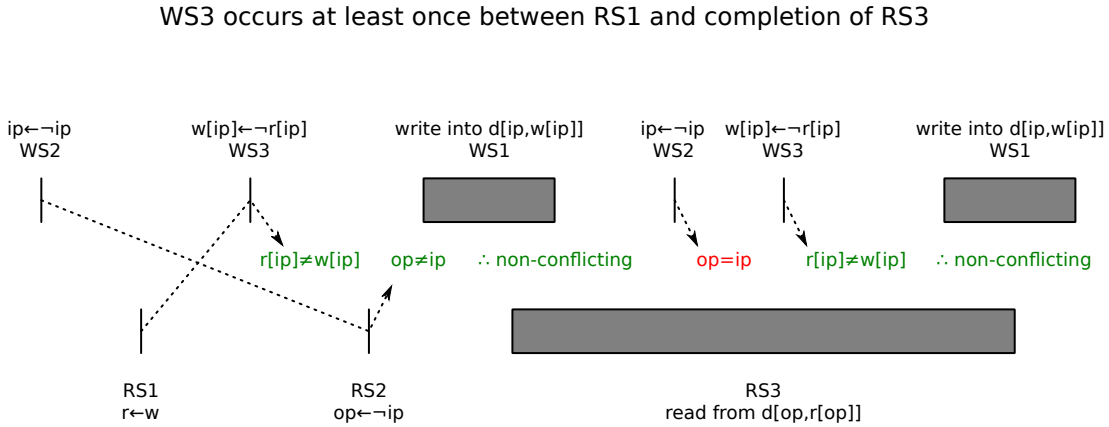

Figure 3.2: W3 occurs at least once between R1 and completion of R3

Lemma 2. *Given an instance of* W3*, if it does occur in the time between the beginning of* R1 *and the completion of* R3*, then the associated* W1 *is non-conflicting because* $r\,[ip] \neq w\,[ip]$*,* $op \neq ip$ *or it does not overlap with a conflicting* R3*.*

Proof. Suppose there is a W1 which potentially conflicts with R3, and the associated W3 is preceded by R1, as shown in Figure 3.2. Because R1 controls the value of r, any occurrence of W3 will then set $w\,[ip] \leftarrow \neg r\,[ip]$ so that $w\,[ip] \neq r\,[ip]$ for the associated W1. □

Theorem 1 (Coherency of reader)**.** *If* R3 *may potentially conflict with* W1*, then either* $op \neq ip$ *or* $w\,[ip] \neq r\,[op]$*.*

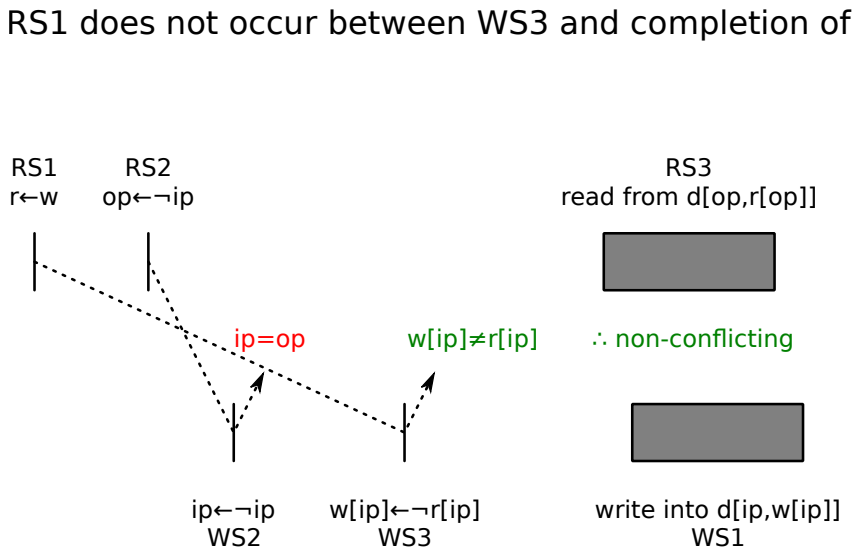

Figure 3.3: R1 does not occur between W3 and completion of W1

Proof. A potential conflict between R3 and W1 means that W1 does not occur strictly before R2.

If the writer has not yet completed its first invocation, and W2 has not occurred at all yet, then $op \neq ip$ because nothing could have changed the value of ip.

Interacting operations R2 and W2 must occur in one order or the other. The same goes for interacting operations R1 and W3. Lemmas 1 and 2 cover the cases. □

Writer

The following lemmas apply to the writer code and are applicable until the writer completes W1.

Lemma 3. *Given an instance of R1, if it does not occur in the time between the beginning of W3 and the completion of W1, then the associated R3 is non-conflicting because $w[ip] \neq r[ip]$, $op \neq ip$ or it does not overlap with a conflicting W1.*

Proof. Suppose there is a R3 which potentially conflicts with W1, but the associated R1 occurred preceding the W3 step, as shown in Figure 3.3. Suppose further that $ip = op$ because R2 preceded W2. Then, because W3 sets $w[ip] \leftarrow \neg r[ip]$ the value of $w[ip] \neq r[ip]$. □

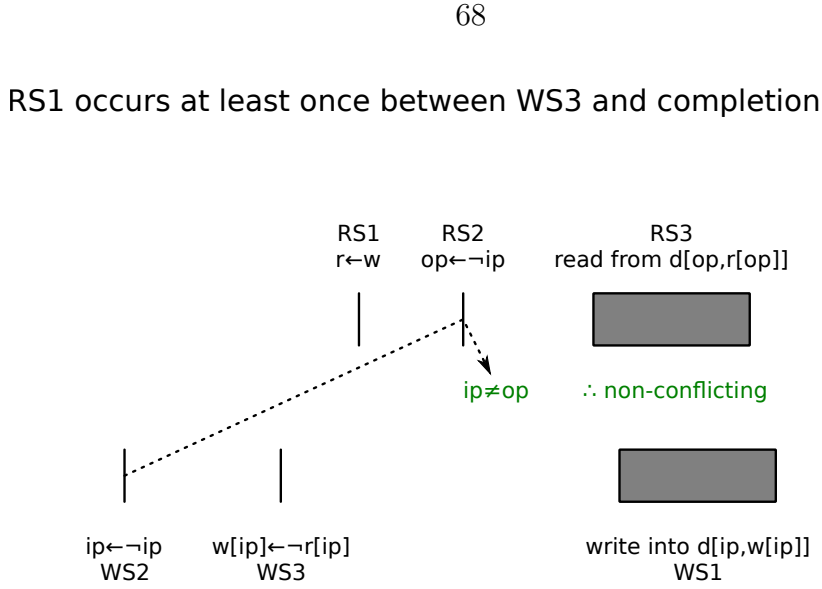

Figure 3.4: R1 occurs at least once between W3 and completion of W1, as well as a case where $op = ip$

Lemma 4. *Given an instance of* R1*, if it does in occur in the time between the beginning of* W3 *and the completion of* W1*, then the associated* R3 *is non-conflicting because* $ip \neq op$ *or it does not overlap with a conflicting* W1*.*

Proof. Suppose there is a R3 which potentially conflicts with W1, and the associated R1 is preceded by W3, as shown in Figure 3.4. By transitivity, that means W2 precedes R2, but then it must be the case that $op \neq ip$. □

Theorem 2 (Coherency of writer)**.** *If* W1 *may potentially conflict with* R3*, then either* $op \neq ip$ *or* $w\,[ip] \neq r\,[op]$*.*

Proof. By theorem 1 we know that the reader will avoid conflicting with the writer. Then we need to show that the writer will avoid conflicting with the reader.

If this is the first time that the writer runs then ip has not yet been modified from its initial value, and the reader must run R2 $op \leftarrow \neg ip$ before reading, so $op \neq ip$. In subsequent runs of the writer code, the question becomes about how W2 and W3 make safe choices for the subsequent W1.

Interacting operations R1 and W3 must occur in one order or the other. Lemmas 3 and 4 cover the cases. □

3.4.5 Single-reader (expanded)

The steps R1, R2, W1 and W2 are, in fact, composed of multiple atomic instructions each. The following diagrams will illustrate that each step has a "linearization" point, when the effect of that step appears to take place, and that the proofs are still valid based on those program points.

3.4.6 Expanded pseudocode

New variables t, u are introduced, temporary local variables used to hold values that are being loaded or stored from memory by atomic operations.

Reader pseudocode

R1a $t \leftarrow w$

R1b $r \leftarrow t$

R2a $t \leftarrow ip$

R2b $op \leftarrow \neg t$

R3 read data from $d\,[op, r\,[op]]$

Writer pseudocode

W1 write data into $d\,[ip, w\,[ip]]$

W2a $u \leftarrow ip$ (see note[1])

W2b $ip \leftarrow \neg u$

W3a $u \leftarrow r\,[ip]$

W3b $w\,[ip] \leftarrow \neg u$

3.4.7 Linearization points

To claim that a program has the property of linearizability is equivalent to the statement that all procedures within have a "linearization point", which is the step when

[1] Modern ARM can do W2a, W2b as a single atomic operation using `LDREX, STREX`

the effect of that procedure appears to take place instantaneously (Herlihy and Wing, 1990).

Of the steps stipulated in proposition 1, here are their linearization points: R1b, R2a, W2b, and W3a. Below are all the diagrams, redone with the expanded pseudocode.

3.4.8 Diagrams

The diagrams are found in Figures 3.5, 3.6, 3.7, and 3.8. The proofs and theorems are the same as in the previous section.

3.4.9 The four-slot mechanism interface

The ATS interface to the four-slot mechanism is shown in Listing 3.2. The view and initialization interface is specified similarly to the IPC memory interface described in Section 3.2, except that this consumes `ipcmem_v` and produces an `either_v` for a `fourslot_v` instead. The `either_v`, in this case, requires that the programmer check the returned status value for `OK` before being allowed to proceed to use the `fourslot_v`, or else if there is failure, then the programmer has to clean-up the `ipcmem_v`. The counterpart function, `fourslot_ipc_free`, consumes a `fourslot_v` and returns an `ipcmem_v`. Finally, the read and write functions are templates that allow the communicated item to be passed by-value.

3.5 The multi-reader mechanism

This section continues the discussion from the previous section and expands the definitions to accommodate the presence of multiple programs reading from the ACM at the same time.

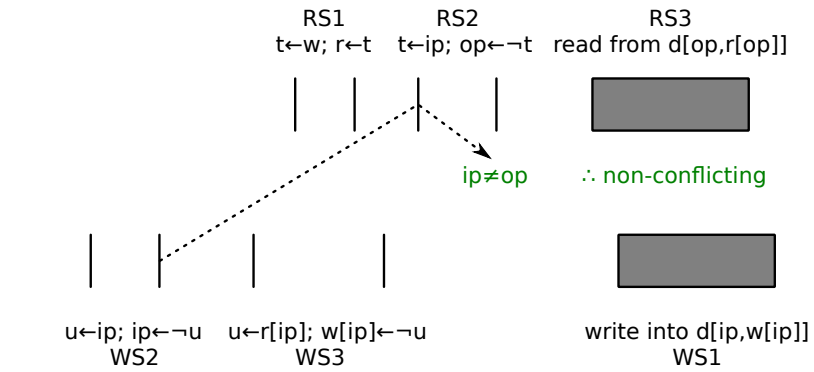

Figure 3.5: R1 occurs at least once between W3 and completion of W1, as well as a case where $op = ip$

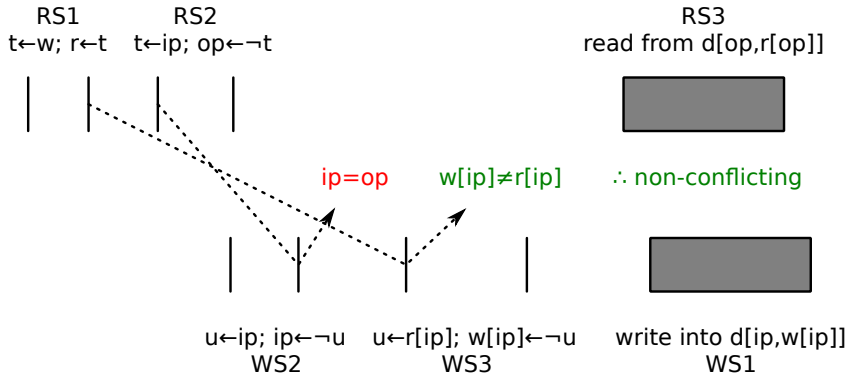

Figure 3.6: R1 does not occur between W3 and completion of W1

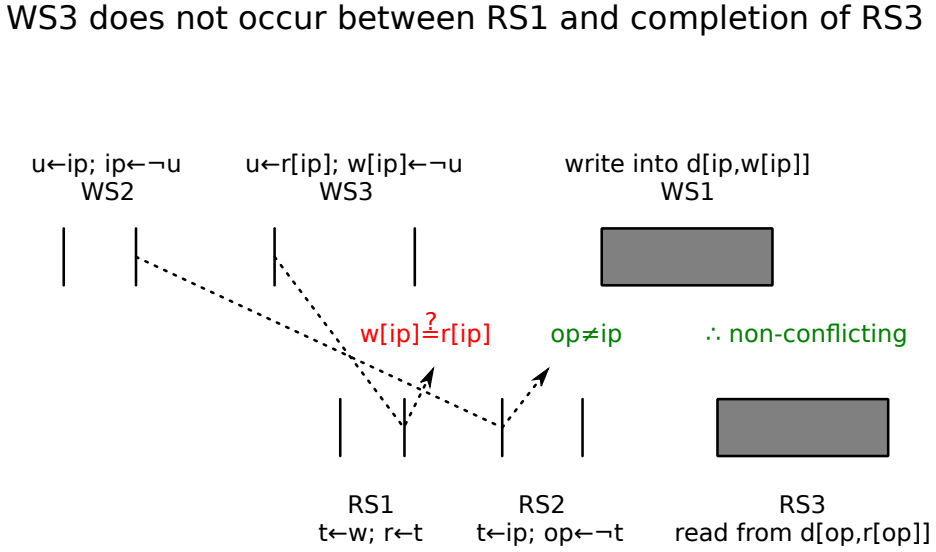

Figure 3.7: W3 does not occur between R1 and completion of R3

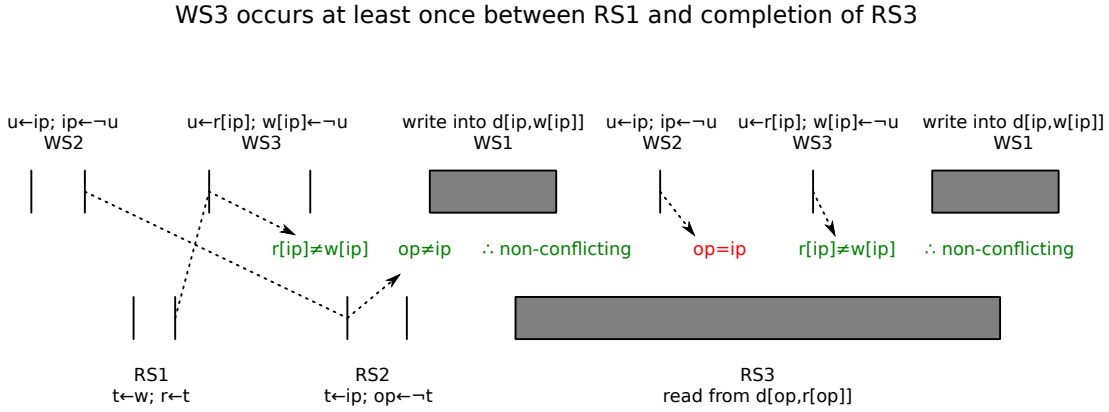

Figure 3.8: W3 occurs at least once between R1 and completion of R3

```
absview fourslot_v (l: addr, n: int, a: t@ype, w: bool)

fun{a:t@ype} fourslot_ipc_writer_init {l: agz} {pages: nat} (
    ! ipcmem_v (l, pages) >> either_v (ipcmem_v (l, pages),
                                       fourslot_v (l, pages, a, true),
                                       s == 0) |
    ptr l, int pages
  ): #[s:int] status s

fun{a:t@ype} fourslot_ipc_reader_init {l: agz} {pages: nat} (
    ! ipcmem_v (l, pages) >> either_v (ipcmem_v (l, pages),
                                       fourslot_v (l, pages, a, false),
                                       s == 0) |
    ptr l, int pages
  ): #[s:int] status s

prfun fourslot_ipc_free {l: addr} {pages: nat} {a: t@ype} {w: bool} (
  fourslot_v (l, pages, a, w)
): ipcmem_v (l, pages)

fun{a:t@ype} fourslot_read {l: addr} {n: nat} (
  ! fourslot_v (l, n, a, false) | fs: ptr (l)
): a

fun{a:t@ype} fourslot_write {l: addr} {n: nat} (
  ! fourslot_v (l, n, a, true) | fs: ptr (l), item: a
): void
```

Listing 3.2: The four-slot mechanism interface

3.5.1 Definitions

Definition 5 (Family of readers)**.** *Suppose there are n readers. Each reader is denoted by R^i and its steps by $\mathsf{R}^i j$, where $0 \leq i < n$. The various variables are defined as follows:*

- n is the number of readers.
- $ip, op_0, \ldots, op_{n-1} \in \{0, 1\}$
- the number of slots $|d| = 2n + 2$
- $|w| = |r_0| = \ldots = |r_{n-1}| = 2$
- $w[0], r_0[0], \ldots, r_{n-1}[0] \in \{0, \ldots, n-1\}$
- $w[1], r_0[1], \ldots, r_{n-1}[1] \in \{0, \ldots, n-1\}$

Informally, the main difference here is that we have introduced a family of variables named op_i, one for each reader, and a family of two-element arrays named r_i, one for each reader. In addition, the arrays are no longer bit-arrays but rather have each element contain values from 0 up to $n-1$. And of course, it is no longer merely a "four slot" array but a $2n+2$ slot array.

Definition 6 (Generalized negation)**.** *$v' = \neg(v_0, \ldots, v_{n-2})$ is an operation with $n-1$ operands. The result of this operation is defined to be unequal to any of the operands: for all $i < n-1$, it is the case that $v' \neq v_i$. If there is more than one allowable result value, the choice is arbitrary. Examples: suppose $n = 3$ then $\neg(0, 1) = 2$ and $\neg(0, 0)$ can be 1 or 2.*

Property 2. *Steps $\mathsf{W}2$, $\mathsf{W}3$, $\mathsf{R}^i 1$, and $\mathsf{R}^i 2$ are all linearizable.*

3.5.2 Pseudocode

Reader pseudocode

$\mathsf{R}^i 1$ $r_i \leftarrow w$

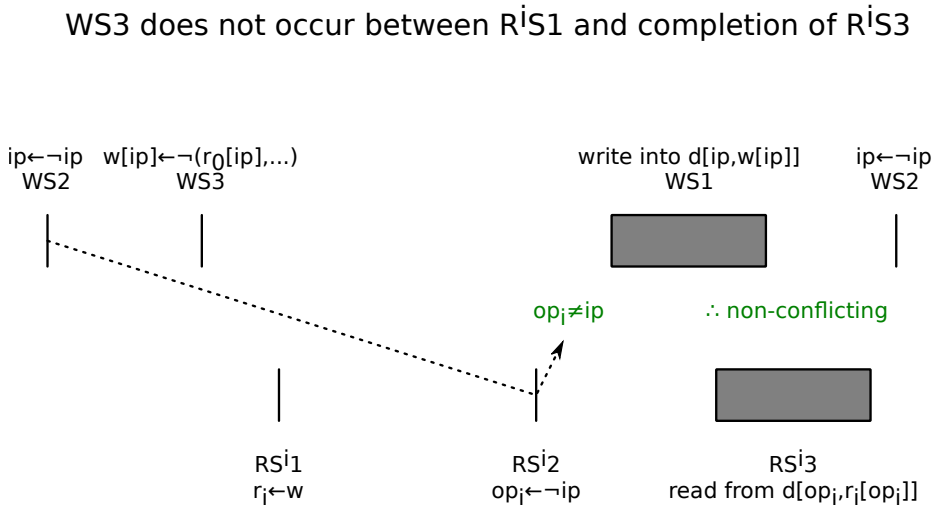

Figure 3.9: W3 does not occur between R^i1 and completion of R^i3

R^i2 $op_i \leftarrow \neg ip$

R^i3 read data from $d\left[op_i, r_i\left[op_i\right]\right]$

Writer pseudocode

W1 write data into $d\left[ip, w\left[ip\right]\right]$

W2 $ip \leftarrow \neg ip$

W3 $w\left[ip\right] \leftarrow \neg\left(r_0\left[ip\right], \ldots, r_{n-1}\left[ip\right]\right)$

3.5.3 Proofs

Reader

The following lemmas apply to the code of reader R^i and are applicable until that reader completes R^i3.

Lemma 5. *Given an instance of* W3*, if it does not occur in the time between the beginning of* R^i1 *and the completion of* R^i3*, then the associated* W1 *is non-conflicting because* $ip \neq op_i$ *or it does not overlap with a conflicting* R^i3*.*

Proof. Suppose there is a W1 which potentially conflicts with R^i3, but the associated W3 operation occurred preceding the R^i1, as shown in Figure 3.9. Then by transitivity,

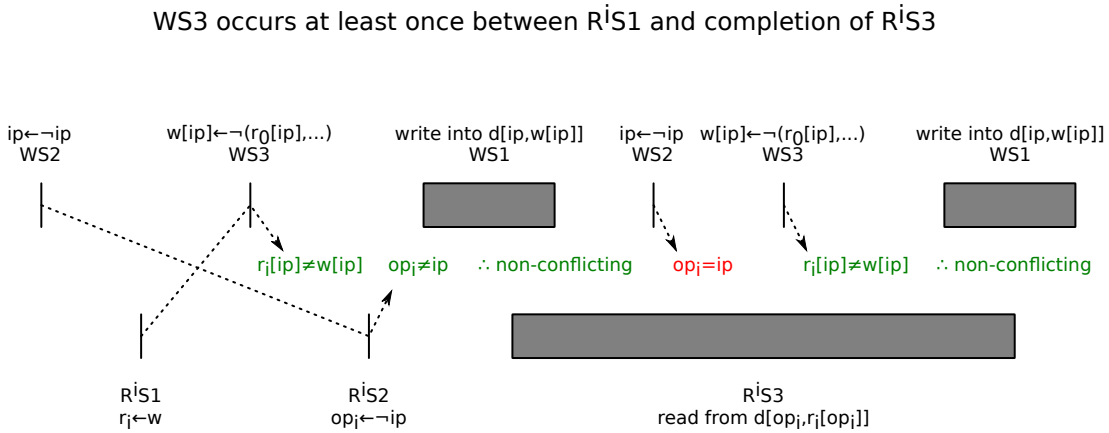

Figure 3.10: W3 occurs at least once between R^{i}1 and completion of R^{i}3, as well as a case where $ip = op_i$

it must be the case that the associated W2 preceded R^{i}2, and therefore it is safe to say that $ip \neq op_i$ because R^{i}2 sets $op_i \leftarrow \neg ip$. □

Lemma 6. *Given an instance of* W3*, if it does occur in the time between the beginning of* R^{i}1 *and the completion of* R^{i}3*, then the associated* W1 *is non-conflicting because* $w[ip] \neq r_i[ip]$*,* $ip \neq op_i$ *or it does not overlap with a conflicting* R^{i}3*.*

Proof. Suppose there is a W1 which potentially conflicts with R^{i}3, and the associated W3 is preceded by R^{i}1, as shown in Figure 3.10. Because R^{i}1 controls the value of r_i, any occurrence of W3 will then set $w[ip] \leftarrow \neg r_i[ip]$ so that $w[ip] \neq r_i[ip]$ for the associated W1. □

Theorem 3 (Coherency of reader)**.** *If* R^{i}3 *may potentially conflict with* W1*, then either* $ip \neq op_i$ *or* $w[ip] \neq r_i[op]$*.*

Proof. A potential conflict between R^{i}3 and W1 means that W1 does not occur strictly before R^{i}2.

If the writer has not yet completed its first invocation, and W2 has not occurred at all yet, then $ip \neq op_i$ because nothing could have changed the value of ip.

Interacting operations R^{i}2 and W2 must occur in one order or the other. The same goes for interacting operations R^{i}1 and W3. Lemmas 5 and 6 cover the cases. □

Writer

The following lemmas apply to the writer code and are applicable until the writer completes W1.

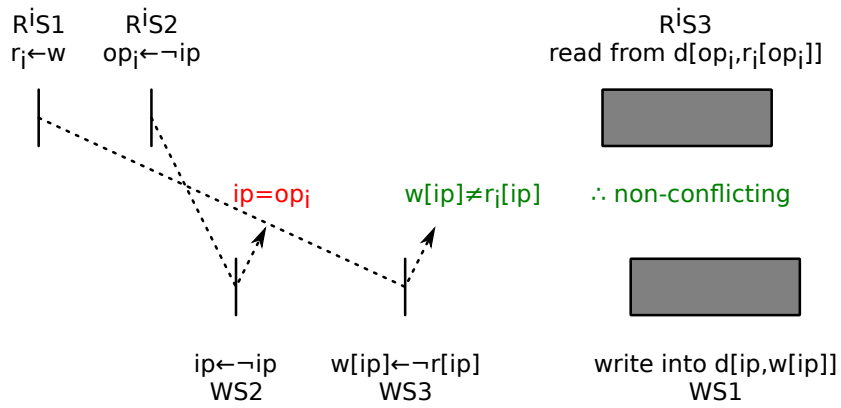

Figure 3.11: R^i1 does not occur between W3 and completion of W1

Lemma 7. *Given an instance of* R^i1*, if it does not occur in the time between the beginning of* W3 *and the completion of* W1*, then the associated* R^i3 *is non-conflicting because* $w\,[ip] \neq r_i\,[ip]$*,* $ip \neq op_i$ *or it does not overlap with a conflicting* W1*.*

Proof. Suppose there is a R^i3 which potentially conflicts with W1, but the associated R^i1 occurred preceding the W3 step, as shown in Figure 3.11. Suppose further that $ip = op_i$ because R^i2 preceded W2. Then, because W3 sets $w\,[ip] \leftarrow \neg r_i\,[ip]$ the value of $w\,[ip] \neq r_i\,[ip]$. □

Lemma 8. *Given an instance of* R^i1*, if it does in occur in the time between the beginning of* W3 *and the completion of* W1*, then the associated* R^i3 *is non-conflicting because* $ip \neq op_i$ *or it does not overlap with a conflicting* W1*.*

Proof. Suppose there is a R^i3 which potentially conflicts with W1, and the associated R^i1 is preceded by W3, as shown in Figure 3.12. By transitivity, that means W2 precedes R^i2, but then it must be the case that $ip \neq op_i$. □

Theorem 4 (Coherency of writer)**.** *For any reader* R^i*, if* W1 *may potentially conflict with* R^i3*, then either* $ip \neq op_i$ *or* $w\,[ip] \neq r_i\,[op]$*.*

Proof. By theorem 3 we know that the reader R^i will avoid conflicting with the writer. Then we need to show that the writer will avoid conflicting with the reader.

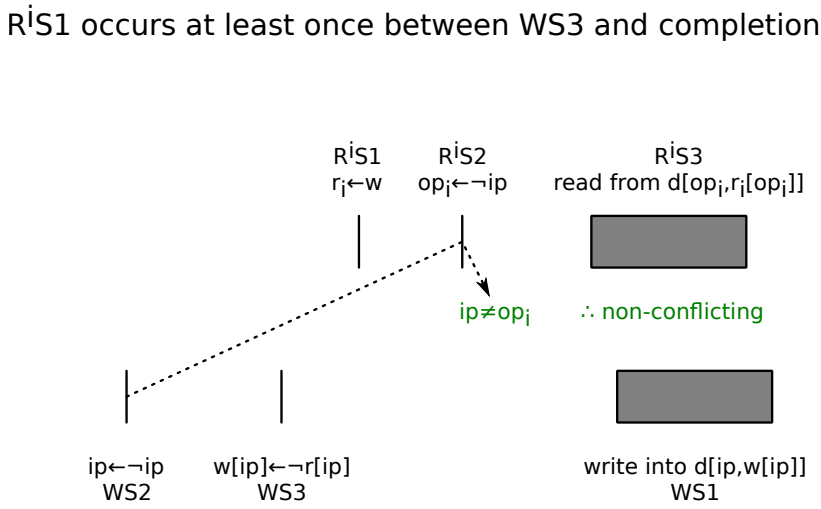

Figure 3.12: R^{i}1 occurs at least once between W3 and completion of W1, as well as a case where $op_i = ip$

If this is the first time that the writer runs then ip has not yet been modified from its initial value, and the reader must run R^{i}2 $op_i \leftarrow \neg ip$ before reading, so $ip \neq op_i$. In subsequent runs of the writer code, the question becomes about how W2 and W3 make safe choices for the subsequent W1.

Interacting operations R^{i}1 and W3 must occur in one order or the other. Lemmas 7 and 8 cover the cases. □

3.5.4 The multi-reader mechanism interface

Listing 3.3 on page 80 shows the multi-reader mechanism interface. This interface is considerably more complicated than the four-slot mechanism for several reasons. One of the issues is that each reader must now carry an associated "index" value, to be passed along whenever invoking the read function. A read index is generated by the initialization function for the reader, and the ATS types ensure that the correct value is always provided to the read function. However, it is another piece of data to be managed by the programmer. Another issue is that the generalization of the mechanism causes a postcondition named `ws3_v` to be generated after every write, and that postcondition must become a precondition of the following write in order

for the proof to hold. An initial `ws3_v` is generated by the initialization function. Although this `ws3_v` parameter must be managed by the programmer, it is erased by the compiler and does not produce any additional overhead. As a result of this `ws3_v`, however, many of the internal indices and assertions are exposed in the types. For example, the condition that `(OP' != IP') || (Rip' != Wip') || (overlap' == false)` is probably not of much interest to application programmers, but this proposition must be encoded for the indices that are used by `ws3_v`, so they must appear in the type. However, other than the read index value, the use of the interface is largely similar to the four-slot mechanism, and the various assertions are all carried along and resolved by the ATS type-checker automatically.

3.5.5 Conclusion

The multi-reader mechanism offers an approach to the single-writer, multi-reader asynchronous communication problem. However, the memory requirement for the multi-reader mechanism scales linearly with the number of readers. Under some circumstances, that poses a difficulty, particularly in cases where the number of readers is not known ahead of time, or under memory space constraints. The following section explores an ACM that can operate within a constant amount of space.

3.6 The fixed-slot mechanism

An analysis of existing ACMs shows that there are several recurring patterns. In order for a writer program to have a free hand, there should always be an extra, unused slot available to be filled, in a scheme reminiscent of double buffering. And for each reader program to operate uninterrupted, there should be a filled slot of which it can take ownership, temporarily. The exact details vary depending on the particular algorithm and whether or not it is taking advantage of hardware atomic operations. But in general, in order to obtain a full expression of the features of

```
absview mslot_v (l: addr, pages: int, a: t@ype, w: bool, i: int)
absview ws3_v (OP: bit, IP: bit, Rip: bit, Wip: bit, overlap: bool)

fun{a:t@ype} multireader_initialize_reader {l: agz} {pages: nat} (
  ipcmem_v (l, n) | ptr (l), int (pages), index: & int? >> int (i)
): #[i: nat] [s: int]
   (either_v (ipcmem_v (l, pages), mslot_v (l, pages, a, false, i), s == 0) |
    status (s))

fun{a:t@ype} multireader_initialize_writer {l: agz} {pages: nat} (
  pf: ipcmem_v (l, n) | p: ptr (l), pages: int (pages)
): [s: int]
   [overlap': bool]
   [OP', IP', Rip', Wip': bit |
       (OP' != IP') || (Rip' != Wip') || (overlap' == false)]
   (either_v (ipcmem_v (l, pages),
              ( mslot_v (l, pages, a, true, 0),
                ws3_v (OP', IP', Rip', Wip', overlap') ),
              s == 0) |
    status (s))

prfun multireader_release {l:addr} {n,i:nat} {a:t@ype} {w:bool} (
  mslot_v (l, n, a, w, i)
): ipcmem_v (l, n)

prfun multireader_release_ws3_v {OP, IP, Rip, Wip: bit} {overlap: bool} (
  ws3_v (OP, IP, Rip, Wip, overlap)
): void

fun{a:t@ype} multireader_read {l: addr} {n, i: nat} (
  ! mslot_v (l, n, a, false, i) | ptr (l), int (i)
): a

fun{a:t@ype} multireader_write
    {l: addr} {n: nat}
    {overlap: bool}
    {OP, IP, Rip, Wip: bit | (OP != IP) || (Rip != Wip) || (overlap == false)} (
  ! mslot_v (l, n, a, true, 0),
  ! ws3_v (OP, IP, Rip, Wip, overlap) >> ws3_v (OP', IP', Rip', Wip', overlap') |
  ms: ptr l, item: a
): #[overlap': bool]
   #[OP', IP', Rip', Wip': bit |
      (OP' != IP') || (Rip' != Wip') || (overlap' == false)]
   void
```

Listing 3.3: The multi-reader mechanism interface

coherency, freshness and concurrency, there is a rising memory requirement as the number of involved programs increases.

If you are willing to relax the strictness of one of coherency, freshness, or concurrency, then it is possible to create a mechanism that can serve any number of readers without increasing the amount of memory devoted to them. For the rest of this discussion, I will focus on the case where I am willing to weaken a feature as a trade off for a fixed memory requirement. In addition, I will focus on the single-writer, multiple-reader design, where the single-writer acts as a broadcaster sending out a piece of reference data to any number of readers willing to participate. After consideration, the feature that I have chosen to weaken is freshness. Coherency is too important for program correctness, and concurrency is fundamental to my entire system's design. However, I have use for an ACM that might occasionally have slightly stale data, if in return it would give me a fixed memory space requirement.

3.6.1 High level functional description and example

The resulting ACM is described by the following principles:

- The single-writer still has its two slots for its double buffering-style scheme.
- The swarm of readers are confined into one or two slots not reserved for the writer.
- If all the readers are working on one slot, then a second slot with fresher data may become available to read.
- As readers finish up with an older piece of data, they are migrated towards the slot with the fresher piece of data.

Therefore, only four slots are absolutely required to handle a single-writer and any number of readers, although more does not hurt. An example of the operation of

these principles is shown in Figure 3.13 on page 83, and the symbols are described in Table 3.1. From left-to-right, and top-to-bottom, the state of the fixed-slot mechanism is shown. In the first row of the figure, you can see a depiction of the swarm of reader threads moving from the slot marked "p" to the slot marked "t". Meanwhile, the writer thread continues working back and forth on the other two slots, leaving the "f" marker behind when it has completed writing into that slot. By the end of the row, all of the readers have completely vacated the "p" slot, therefore the "p" marker is moved to the same slot as the "t" marker. And after that happens, the "t" marker is itself free to advance to the freshest data as indicated by the "f" marker. The remainder of the example continues to show this pattern of markers leading and following the threads.

p	**p**revious target
t	current **t**arget
f	most recently **f**illed
r	**r**eader thread
w	**w**riter thread

Table 3.1: Legend for Figure 3.13

By ensuring that all reader threads may only occupy one of two slots at any time, and by having a single writer thread only, I can show that four data slots is all that is strictly necessary to make this scheme work, no matter the number of readers. The advancement of the "p" and "t" markers occurs in a step-over-step fashion, while the writer thread can flit back and forth between the remaining two slots. Therefore this data structure is different from a circular buffer in which the advancement of the threads through the buffer always proceeds in the same direction. The reason why this ACM compromises freshness is because no reader is allowed to move onto the slot marked "f" until the "p" and "t" markers are placed together. The net effect is that a slow reader thread might hold up the advancement of faster reader threads. Whether this turns out to be a problem in practice remains to be seen.

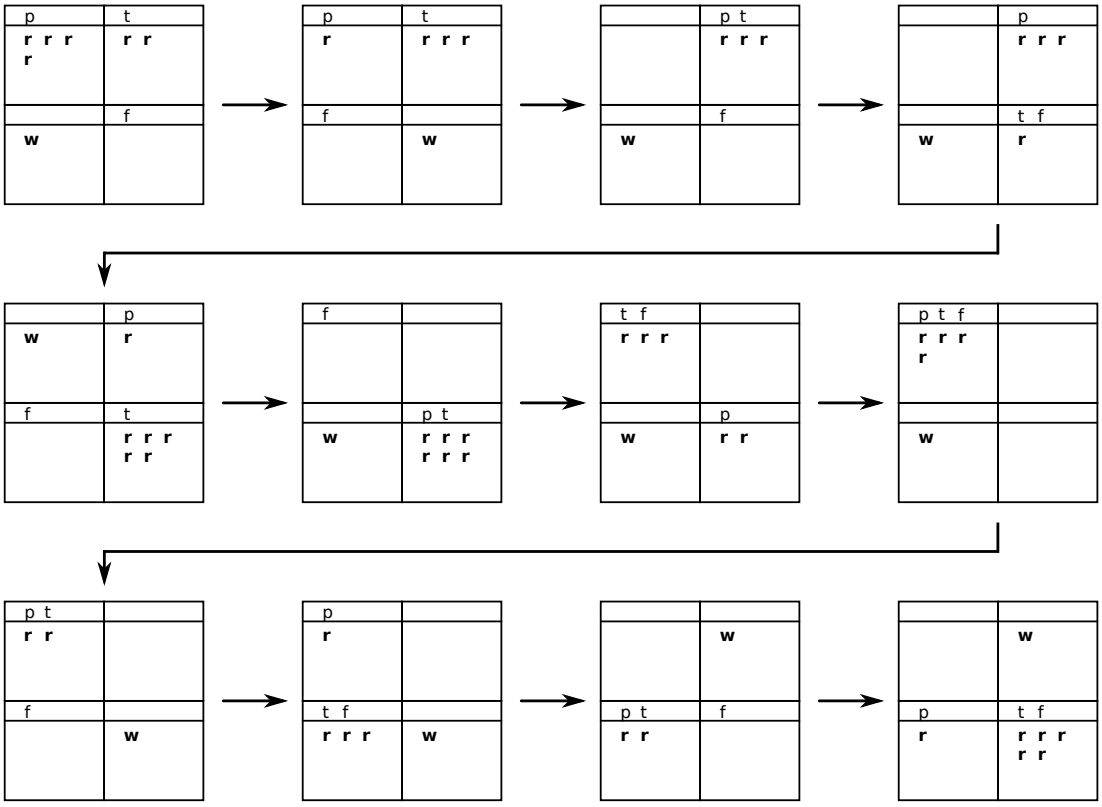

Figure 3.13: Example sequence of steps for a fixed-slot mechanism with four slots and six readers

3.6.2 Pseudo-code

The shared variables are described in Figure 3.14. The *rcount* array stores a count of how many readers are currently working on a particular slot at any time. I call S the "safety" counter and its rationale is described in Section 5.1.1. p, t, f are the markers as described in Table 3.1 and they are represented as indices into the data slot array.

The pseudo-code is laid out in Figure 3.15 and Figure 3.16. The first step of the writer uses the notation $\neg(p, t, f)$ to describe the notion of "generalized negation" as introduced by Simpson (1997a): it produces a number that is not found in the set $\{p, t, f\}$. The writer does not require the use of any atomic operations beyond the ability to load and store integer values from memory coherently, a property called "single-copy atomicity" that will be discussed in Section 3.6.3.

- $rcount\,[i] \in \mathbb{N}$ for slots $i \in \{0, 1, 2, 3\}$,
- $S \in \mathbb{N}$,
- $p, t, f \in \{0, 1, 2, 3\}$ and
- an array sized to fit four slots of data.

Figure 3.14: The shared variable state

W1. $w_i \leftarrow \neg\,(p, t, f)$
W2. write data into slot w_i
W3. $f \leftarrow w_i$

Figure 3.15: Fixed-slot mechanism pseudo-code: writer

The reader is somewhat more complex. Steps R1 and R6 implement the step-over-step logic: when the conditions are right, t chases f, while p chases t. Both R1 and R6 must be implemented as atomic operations, this will be discussed in Section 3.6.3. Steps R2, R3 and R5 are basic increment or decrement operations, and these are trivial to implement atomically using hardware support.

3.6.3 Details of implementation

Atomic operations R1 and R6 are non-trivial and require additional explanation. Architectures in the real world provide various levels of support for hardware atomic instructions, and it often comes in subtly different forms. For the purpose of the Terrier operating system, I will focus on the commonly available ARM architecture (ARM, 2011), and I will discuss the capabilities of the ARM processor in handling atomic instructions. This does not preclude the fixed-slot mechanism from operating on other architectures, but it does require conversion to whatever primitive atomic instructions are available.

R1. `incr` S;
    `if` $p = t$ `then` $\{\ t \leftarrow f\ \}$;
    $r_i \leftarrow t$
R2. `incr` $rcount\,[r_i]$
R3. `decr` S
R4. read data from slot r_i
R5. `decr` $rcount\,[r_i]$
R6. `if` $S = 0$ `and` $rcount\,[p] = 0$ `then` $\{\ p \leftarrow t\ \}$

Figure 3.16: Fixed-slot mechanism pseudo-code: reader

Atomicity and the ARM processor

The 32-bit ARMv7 processor has a few options for implementing atomic operations. One of the most basic is that a 32-bit load or store can be guaranteed to be what is called "single-copy atomic" when operating on normal memory. The gist of this property is that no matter the interleaving, loads will always return a coherent 32-bit value that was previously written, and stores will always manage to save a coherent 32-bit value. The manual states that: *it is impossible for part of the value of the memory location to come from one write operation and another part of the value to come from a different write operation.* The ARM processor also offers single-copy atomicity in 8-bit and 16-bit operations, but I will not take advantage of those.

Past versions of the ARM architecture have offered a now-deprecated `SWP`, or "swap", atomic instruction. However, modern ARM versions have moved onto a much more flexible and sophisticated scheme known popularly as "load-link/store-conditional (LL/SC)" (Jensen et al., 1987) or `LDREX`/`STREX` in ARM parlance. LL/SC is an optimistic scheme that allows an atomic operation to unfold in the time between two instructions: `LDREX` and `STREX`, which stand for "load-exclusive" and "store-exclusive" respectively. The `LDREX` operation retrieves a word from memory and establishes a "monitor" on that memory address. The processor then may go on to execute a series of instructions before finally coming to a `STREX` instruction. The `STREX` instruction is supposed to store a word of memory into the exact same address

as was previously accessed by `LDREX`. However, if anything has disturbed the monitor, then the `STREX` operation will return an error code, and the program is thus informed that the store operation did not take place. The usual procedure is then to loop back and retry the whole operation, starting with the `LDREX` all over again. Various effects can disturb the monitor, especially if another processor comes along and attempts to write to the same memory address that the monitor is watching. An example atomic procedure is shown with pseudo-code in Figure 3.17. Another important property of LL/SC is that the ARM specification guarantees that at least one thread will progress when using `STREX` on the same memory address. This makes it suitable for implementing lock-free algorithms. LL/SC can be used to implement a whole family of atomic operations, such as `Test-And-Set` or `Compare-And-Swap`. And it can be used to do more elaborate operations as well. The ARM manual merely recommends that the number of instructions between `LDREX` and `STREX` be kept to a minimum in order to minimize the chance of disturbing the monitor, and that 128 bytes between the two is probably the upper limit of reasonableness.

Implementing the reader

In short, the use of `LDREX`/`STREX` implies that the code in the monitored section should be free of side-effects except on the specific memory address affected by the `STREX` operation. Therefore, with this fact in mind, I have chosen to implement the S, p, t, f shared state variables as fields in a single 32-bit word, as shown in Table 3.2. Variables p, t, f only require 2 bits of space each, which leaves up to 26 bits worth of room to store the counter S. Technically, this decision restricts the number of

1. $v \leftarrow$ `LDREX` $[address]$
2. $v \leftarrow v + 1$
3. $r \leftarrow$ `STREX` $v, [address]$
4. `if` $r \neq 0$ `then goto` 1

Figure 3.17: Atomic increment using `LDREX`/`STREX`

bits:	2	2	2	26	
32	p	t	f	S	0

Table 3.2: The bit layout of the 32-bit word

simultaneous readers to 2^{26} but I feel that this limitation is more than sufficient for any conceivable system.

The implications of this layout are that incrementing and decrementing S can be done the usual way, with compiler primitives that expand into assembly code much like shown in Figure 3.17. Accessing the shared state variables p, t, f does require masking and shifting, but the ARM processor is quite adept at that task, as many instructions can invoke the inline "barrel shifter" functionality at no additional cost. In any case, having access to all state variables simultaneously is advantageous for implementing operations such as R1 and W1.

Sample pseudo-code for the atomic operations R1 and R6 are shown in Figures 3.18 and 3.19. The pseudo-code largely reflects the actual ARM assembly code but more comprehensible names are substituted for various operations:

- GetBits (r, off, len) and SetBits (r, off, len) perform shifting and masking operations to get or set the bit-fields as specified by an offset and a length.
- PSHIFT, TSHIFT, FSHIFT, and SLENGTH are constants intended to encode the layout of the 32-bit word as described in Table 3.2.
- Pseudo-registers with a meaningful label, such as r_t, are used instead of actual registers.
- `if ··· then goto` is used instead of the normal comparison, branch or conditional execution codes.

Notably, the assembly code is no more than about a dozen instructions, which is far below the limits suggested by the ARM manual. The code largely consists

R1. `incr` S; `if` $p = t$ `then` $\{\ t \leftarrow f\ \}$; $r_i \leftarrow t$

1. $r_w \leftarrow$ `LDREX` $[address]$
2. $r_w \leftarrow r_w + 1$
3. $r_p \leftarrow \mathsf{GetBits}\,(r_w, \mathsf{PSHIFT}, 2)$
4. $r_t \leftarrow \mathsf{GetBits}\,(r_w, \mathsf{TSHIFT}, 2)$
5. `if` $r_p \neq r_t$ `then goto` 8
6. $r_f \leftarrow \mathsf{GetBits}\,(r_w, \mathsf{FSHIFT}, 2)$
7. $\mathsf{SetBits}\,(r_w, \mathsf{TSHIFT}, 2) \leftarrow r_f$
8. $r_c \leftarrow$ `STREX` $r_w, [address]$
9. `if` $r_c \neq 0$ `then goto` 1
10. $r_i \leftarrow \mathsf{GetBits}\,(r_w, \mathsf{TSHIFT}, 2)$

Figure 3.18: Assembly pseudo-code for atomic operation R1

R6. `if` $S = 0$ `and` $rcount\,[p] = 0$ `then` $\{\ p \leftarrow t\ \}$

1. $r_w \leftarrow$ `LDREX` $[address]$
2. $r_S \leftarrow \mathsf{GetBits}\,(r_w, 0, \mathsf{SLENGTH})$
3. `if` $r_S \neq 0$ `then goto` 9
4. $r_p \leftarrow \mathsf{GetBits}\,(r_w, \mathsf{PSHIFT}, 2)$
5. $r_n \leftarrow$ `LOAD` $[address\ of\ rcount\,[r_p]]$
6. `if` $r_n \neq 0$ `then goto` 9
7. $r_t \leftarrow \mathsf{GetBits}\,(r_w, \mathsf{TSHIFT}, 2)$
8. $\mathsf{SetBits}\,(r_w, \mathsf{PSHIFT}, 2) \leftarrow r_t$
9. $r_c \leftarrow$ `STREX` $r_w, [address]$
10. `if` $r_c \neq 0$ `then goto` 1

Figure 3.19: Assembly pseudo-code for atomic operation R6

of shifting and masking operations performed on a single word, with no side-effects outside of that. R6 does reach out to perform a memory load from an address not under the exclusive monitor, but that is also acceptable under the ARM specification: so long as the access is to normal memory.

Implementation of the writer

The writer performs fewer steps and each of them is less complex. For W1, a more generalized negation operation might invoke the `CLZ` instruction (count-leading-zeroes), but with only four possible result values, the most efficient implementation here in-

volves simply setting and testing a few bits in a register. W3 is simply a couple of shifting and masking operations to set the value of f, wrapped in the usual `LDREX`/`STREX` pair. For both the reader's R4 and the writer's W2, the actual reading and writing of data can be left to the compiler's most efficient version of `memcpy`.

Given all of these assembly sections, I can examine them and their context to figure out reasonable pre-conditions and post-conditions. And I know that `LDREX` and `STREX` will always allow at least one thread to progress. So, the following question is: can I put this all together and show that they work together in sequence to give me the desired property of coherency? That is the topic of Chapter 5, and for the fixed-slot mechanism specifically, in Section 5.1.

3.6.4 The fixed-slot mechanism interface

Types and initialization

Some simplifications have been made in the discussion of the fixed-slot mechanism thus far in this chapter, as well as in Section 5.1. These simplifications shorten the description for the purpose of discussing the workings of the mechanism, without affecting its correctness. In this section, I will introduce the true interface for the fixed-slot mechanism, which adds annotations that are useful to the application programmer. In Listing 3.4 you can see the true viewtype definitions for the fixed-slot mechanism. These types are indexed by two parameters of interest to users of the interface: the type of data being communicated by the mechanism, and the direction, encoded as a boolean value (at the type level) that is true iff this is the writer side. Two convenient aliases are also provided, one for writers, and one for readers. The definition of the `fixedslot` type is omitted here, since it is an existential type that is hiding details only of interest to the implementation.

Listing 3.5 shows the set of initialization and free functions available. They make use of the IPC memory interface described in Section 3.2. There are separate functions

```
vtypedef fixedslot (a: t@ype, wr: bool)
vtypedef fixedslotw (a: t@ype) = fixedslot (a, true)
vtypedef fixedslotr (a: t@ype) = fixedslot (a, false)
```

Listing 3.4: Types for the fixed-slot mechanism

```
fun fixedslot_initialize_reader {a: t@ype} {l: addr} {n: nat} (
  ipcmem_v (l, n) | ptr l, int n
): fixedslotr a

fun fixedslot_initialize_writer {a: t@ype} {l: addr} {n: nat} (
  ipcmem_v (l, n) | ptr l, int n
): fixedslotw a

fun fixedslot_free {a: t@ype} {wr: bool} (
  fixedslot (a, wr)
): [l: addr] [n: nat] (ipcmem_v (l, n) | ptr l)
```

Listing 3.5: Initialization for the fixed-slot mechanism

for initializing readers and writers, but only one is needed for releasing the memory back to the IPC memory pool. These functions cannot fail so no special error-handling is needed.

Basic interface

For reading and writing of small values that are not awkward to pass by value, a simple interface is provided as shown in Listing 3.6. As will be common to all the following interface functions, the viewtype for the fixed-slot mechanism is shown to evolve even though there are no apparent changes at this level, e.g. `! fixedslotr a >> _`. In fact, since this viewtype has existential variables, there are changes behind the scenes. ATS provides a convenient syntax using the underscore that allows us to express that although the type has remained the same at the lexical level, there are changes going on that are hidden from us: and therefore when the function returns, it means that the value has been updated in some fashion.

```
fun{a:t@ype} fixedslot_read (! fixedslotr a >> _): a
fun{a:t@ype} fixedslot_write (! fixedslotw a >> _, a): void
```

Listing 3.6: Basic interface for the fixed-slot mechanism

```
fun fixedslot_readptr {a:t@ype} {l: agz} (
  !a @ l | ! fixedslotr a >> _, ptr l, size_t (sizeof a)
): void

fun fixedslot_writeptr {a:t@ype} {l: agz} (
  !a @ l | ! fixedslotw a >> _, ptr l, size_t (sizeof a)
): void
```

Listing 3.7: Pointer-based interface for the fixed-slot mechanism

Pointer interface

For larger types, such as arrays, it is not such a good idea to pass by value. A reference interface could have been provided but I have opted to express it as a pointer-with-views instead. In Listing 3.7, for this interface, the programmer must provide a view that proves the right to access the piece of memory for the given pointer, as well as the correct size value for the type. If these are not provided correctly, then type-checking shall fail.

Higher-order function interface

Perhaps the most intriguing opportunity for an ATS interface is the potential of using higher-order functions to express code in better, possibly even more more efficient ways. Listing 3.8 shows an interface that allows a programmer to pass along a function that takes a reference to the communicated data and then returns some other value. This may be useful, for example, if you only wish to examine a small portion of the incoming data, and therefore doing a full copy into a buffer is unnecessary work.

A working example is provided in Listing 3.9. For this use, the buffer is quite large (1600 bytes), but for the time being I am only interested in the first four bytes. The combination of `var`, `lam@`, and `=<clo1>` has allowed me to create a downward-only

```
fun{b:t@ype} fixedslot_readfn {a:t@ype} (
  ! fixedslotr a >> _, & ((& INV(a)) -<clo1> b), size_t (sizeof a)
): b
```

Listing 3.8: Higher-order function interface for the fixed-slot mechanism

```
fixedslot_readfn<uint> (input, f, BUFSIZE) where {
  var f = lam@ (rbuf: & buf_t): uint =<clo1> rbuf[0]
}
```

Listing 3.9: Higher-order function example for the fixed-slot mechanism

closure that is allocated upon the stack. It is a function that can be passed as a value, but it cannot be returned, in order to protect the dynamic extent of the closure. But I only need it to survive long enough to finish running the `fixedslot_readfn` function. In this case, instead of reading 1600 bytes into a separate buffer, I simply dereference and index into the `buf_t`, which happens to be defined as an array of 400 `uints`. The `fixedslot_readfn` function operates the mechanism far enough to obtain the proper state for reading, then calls my function, and then cleans up afterwards. It is also fully parametric in that it returns the value that my function returned. In this way, I have implemented a function that is both simple and efficient, and it manages to capture this essence of functional programming in a systems setting.

3.6.5 Performance

I conducted several experiments to find out whether the atomic operations were practical in the presence of several competing threads: I believe that the results are reasonable. Those results are shown in Table 3.3 from a testbed with a Cortex-A9 MPCore ARM processor running the Terrier OS. One writer and three readers were tested under various rate-monotonic static priority scheduling scenarios. The number of cycles required to complete the atomic operation R1 was measured for each invoca-

tion, starting from before `LDREX` and recording the cycle count after `STREX` completed successfully. Each run could result in over a hundred thousand invocations of R1. The total number of cycles and invocations was then added up and divided to give an average for each testing scenario.

As you can see in the table, certain scheduling scenarios result in higher average cycle counts for R1. I suspect that this is due to increased contention on the exclusive monitor, and unfortunate timing of context switching away from the running process. The worst case came when all programs shared the same capacity and period values, and it did not seem to matter whether or not they were split across the different CPUs. One interesting note from the tests is not shown in the table: when R1 was able to run without any interference, it could finish in as little as 27 cycles.

A set of larger experiments were conducted as well, mostly involving interaction with the USB interface. These experiments happened in conjunction with the experiments described in Section 4.1.3, which may need to be consulted in order to understand the full context. In short: a USB network interface card is being used to send and receive Ethernet frames, and the μIP library is providing a basic network

	test1	*test2*	*test3*	*test4*	*test5*
writer					
capacity	1	1	1	1	1
period	4	4	7	5	7
affinity	cpu0	cpu0	cpu0	cpu0	cpu0
reader0					
capacity	1	2	1	1	1
period	9	9	7	7	7
affinity	cpu1	cpu1	cpu1	cpu1	cpu0
reader1					
capacity	1	3	1	1	1
period	9	9	7	11	7
affinity	cpu1	cpu1	cpu1	cpu1	cpu0
reader3					
capacity	1	4	1	1	1
period	9	9	7	13	7
affinity	cpu0	cpu0	cpu0	cpu0	cpu0
R1 avg cycles	86.8	74.3	124.9	87.7	122.0

Table 3.3: Average cycle counts to run step R1 under various scheduling scenarios

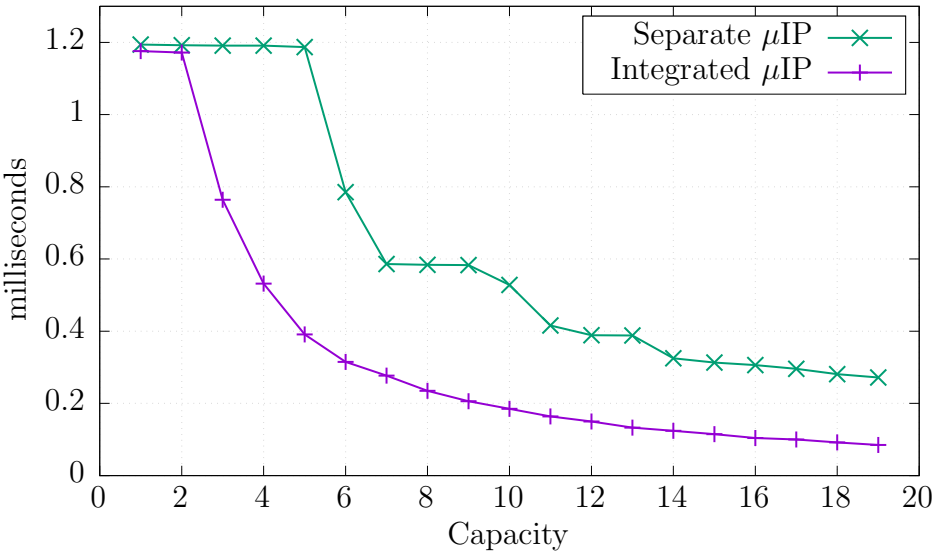

Figure 3.20: Experiment with separate μIP process of varying capacity

stack.

Table 3.4 shows the basic setup of the processes on the CPUs. Capacity and Period units are given in multiples of 1/16384 of a second, approximately 61 μseconds each. C_1 and C_2 were set and varied in various ways: when not varied, the default C_1 or C_2 chosen was 19. The NIC1 process used the fixed-slot mechanism to pass received packets to the μIP process, and the μIP process used the fixed-slot mechanism to send packets back to the NIC1 for transmission. Therefore, each ICMP ping must go twice between processes. This style of programming more closely resembles that of a microkernel; it is not generally recommended for Terrier. However, it does provide a good example of an IPC task that must take place within a timely manner.

	CPU0		CPU1
	EHCI	μIP	NIC1
Capacity	16	C_1	C_2
Period	320	20	20

Table 3.4: Basic scheduling setup for the experiments

Figure 3.20 compares the performance of a system with a separate μIP process

that must rely on the fixed-slot mechanism for communication against a system where the μIP stack is integrated into the NIC1 process. The capacity of the μIP process, C_2, is varied from 1 to 20. As expected, the separate μIP system performs worse

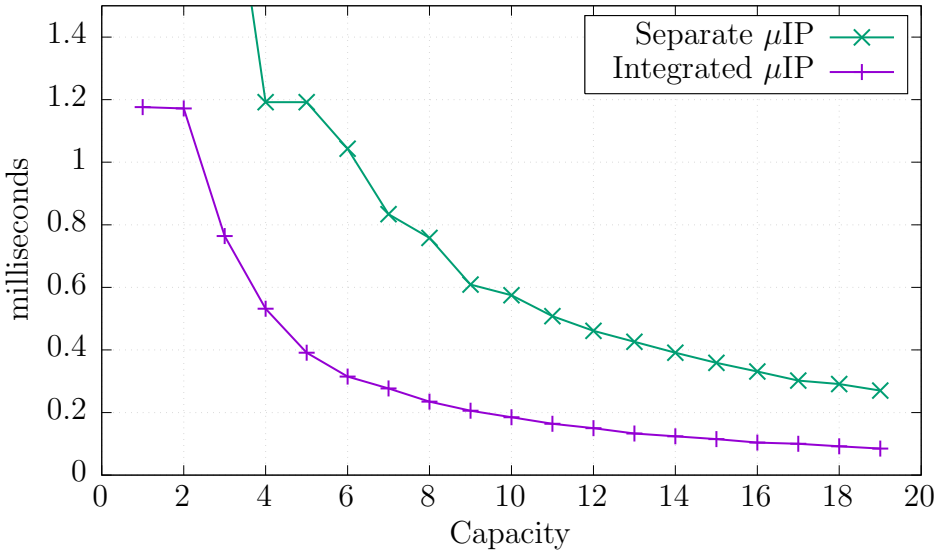

Figure 3.21: Experiment varying capacity of NIC1 with separate μIP process

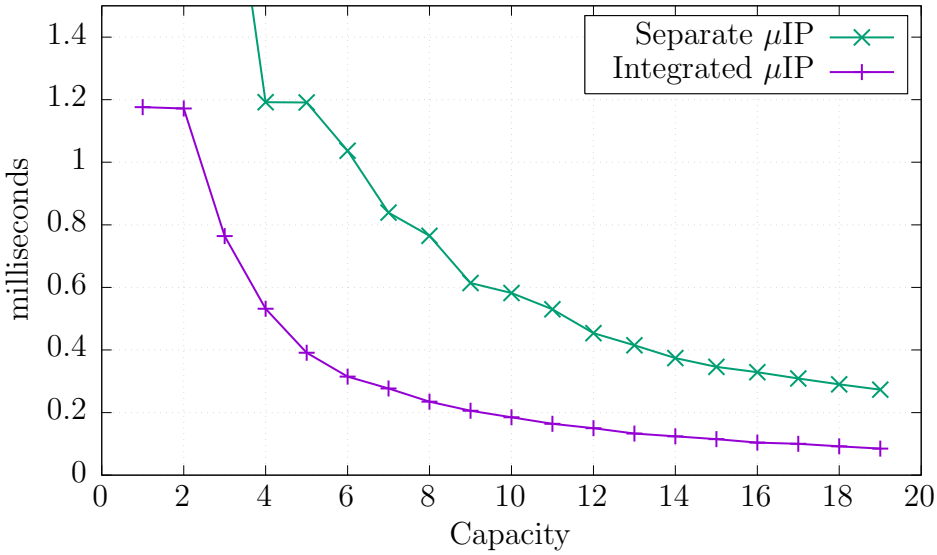

Figure 3.22: Experiment varying capacity of both NIC1 and separate μIP process

on this responsiveness test than the integrated system; it has to communicate two additional times, after all. However, the overall difference is about 200 μsec, and it still outperforms Angstrom Linux as long as $C_2 > 13$. Altogether, this suggests that each operation of IPC had overall overhead at about 100 μsec: to transfer a 1600-byte buffer across processors via the asynchronous fixed-slot mechanism, in an experiment where the two processes were allowed to operate completely independently of each other.

Next, a similar experiment was conducted but instead of varying C_2, I varied C_1, the NIC1 capacity. Those results are shown in Figure 3.21. And in Figure 3.22 I varied both C_1 and C_2 simultaneously. In all cases, the difference in response time between the integrated and the separate μIP processes are present in approximately the same way, although the performance in the extremely low capacity case varied significantly.

Chapter 4

Memory Protection using Types

One of the primary goals of Terrier is to gain the advantage of memory protection by employing the use of type system-level reasoning instead of safety checks at run-time. The following sections describe how that philosophy was applied to various subsystems within the operating system.

4.1 The USB interface

The low-level process of setting up the USB host controller, with the Enhanced Host Controller Interface (EHCI), is described in Section 2.3.4. The initialization takes place under the `ehci` module, which also publishes a shared, inter-process piece of memory that allows other programs to perform USB operations directly upon the internal structures of the host controller driver without any additional inter-process communication or kernel-privilege level switching. The interface to that shared memory is specified by `userlib/usb.sats`. It is also expected that most users of the USB interface will want to take advantage of the prewritten USB data structure manipulation library that is described in the same file. This provided library has functions with types that help ensure the proper steps are followed when preparing, executing and analyzing USB transfers. The following sections describe an example and the provided interface for executing USB transfers asynchronously.

4.1.1 An example use of the interface

Listing 4.1 shows the implementation of a function that invokes the `ClearFeature` command on a USB endpoint to clear a halted endpoint, using a standard USB control transfer.

```
fun usb_clear_endpoint {i: nat} (
    dev: ! usb_device_vt (i), endpt: int
  ): [s: int] status (s) =
  usb_with_urb (dev, 0, 0, f) where {
  var f = lam@ (
             dev: ! usb_device_vt (i),
             urb: ! urb_vt (i, 0, false)
           ): [s: int] status (s) =<clo1>
    let
      val (xfer_v | s) =
          urb_begin_control_nodata (
            urb, make_RequestType (HostToDevice, Standard, Endpoint),
            make_Request ClearFeature,
            ENDPOINT_HALT, endpt
          )
    in
      if s = OK then begin
        urb_wait_while_active (xfer_v | urb);
        urb_transfer_completed (xfer_v | urb);
        urb_detach_and_free urb
      end else begin
        urb_transfer_completed (xfer_v | urb);
        urb_detach_and_free_ urb;
        s
      end
    end
}
```

Listing 4.1: `usb_clear_endpoint`

I make use of the higher-order function `usb_with_urb` to neatly bracket the use of the USB Request Block (URB) with the appropriate resource allocation and release operations. ATS makes it possible to use higher-order functions such as `usb_with_urb` in a low-level setting by allocating the closure on the stack. This is accomplished by the syntax shown in the listing, where `var f` is defined as a function using the `lam@` keyword and a linear closure with effects using the `=<clo1>` syntax after the type signature. Such a stack-allocated closure may be used in downward function calls but cannot be returned up the stack, and the type system prevents that from happening.

The viewtype `urb_vt` representing the URB is an abstract stand-in that is concretely represented behind the scenes as a standard EHCI queue head structure. Several such URBs are prepared by the `ehci` module and kept in reserve, so that

they can be allocated at will for each device. Statically, the abstract viewtype `urb_vt` keeps track of the index of the device, the number of Transfer Descriptors currently attached, and whether or not they are active as far as the program knows:

```
absvtype urb_vt (i: int, nTDs: int, active: bool)
```

The function `usb_begin_control_nodata` begins a type of USB control transfer that has no DATA stage, only the SETUP and STATUS stages. The SETUP stage of any USB control transfer is given in a standard USB device request format that has a number of fields defined by the USB specification. They are flexible and intended for many different uses depending on the specific request being made. Here they have been encoded as such:

- `usb_RequestType_t` is created by a `make_RequestType` function (see Listing 4.8) that expects three values defined with algebraic datatypes, but those are essentially translated by ATS into enumerated integer types, so there is no overhead.
- The same goes for `usb_Request_t` and `make_Request`.
- The next value is known as the `wValue` in USB parlance and in this particular case, it specifies the particular feature to be cleared, namely the `ENDPOINT_HALT`.
- The final parameter is the `wIndex` and for this particular request it is defined to be the endpoint that should be cleared. Notice that the variable `endpt` is part of the closure.

The USB interface provided by the `usb.sats` library has a set of functions, such as `urb_begin_control_nodata`, that set up the data structures and initiate the transfer but then return and expect the clean-up to be handled by the caller. With ATS types we can easily make specifications that require the caller to clean up properly.

In this case, the function that begins the control transfer returns dynamically a single integer value that specifies the status of the transfer. It also returns a static

```
! urb_vt (i, 0, false) >> urb_vt (i, nTDs, active) // where
#[nTDs: int | (s == 0 || nTDs == 0) && (s <> 0 || nTDs > 0)] // and
#[active: bool | (s == 0) == active]
```

Listing 4.2: The way `urb_begin_control_nodata` modifies the URB

view, named `xfer_v`, that represents the fact that a transfer might be on-going. A couple of other viewtypes are modified "in-place" as well. The evolution of the URB is shown in Listing 4.2.

These definitions work together: the view representing the URB cannot be fully comprehended until the value of the status `s` is known. The guards encode the following logical statements:

- If `s` is not `OK` then the value of `nTDs` is equal to 0.
- If `s` is `OK` then the value of `nTDs` is greater than 0.
- The value of the boolean static variable named `active` is equal to the result of comparing `s` with the value of `OK`.

After the return from `urb_begin_control_nodata` we must deal with the URB because it is now possibly active, with a chain of allocated TDs to manage. So, with the ATS type system, we are required to deal with the value of `urb_vt` in some way because it has a linear type. But we cannot choose the functions to deal with it until we know whether or not the status is `OK`. In this way, we force the application programmer to do the proper error-checking and handling, and this is all handled statically during type-checking.

Let us consider the `else` branch first: the function `urb_transfer_completed` is defined to require that the URB object it operates upon is not active. Thanks to the `if`-statement we know that is the case.

Finally, we must take responsibility for the chain of transfer descriptors that is attached to the URB. Since they represent allocated memory, they must be managed.

```
absvtype usb_device_vt (i: int)

fun usb_acquire_device {i: nat} (int i): usb_device_vt (i)

fun usb_release_device {i: nat} (usb_device_vt (i)): void
```

Listing 4.3: USB device acquisition and release

Again, the types stipulate that we must return a URB with zero TDs attached, so ATS will not let us off the hook until we do something about the TDs. The function `urb_detach_and_free_` is a helper function that unhooks the TDs from the USB device and also deallocates all of them. It returns `void`, so we simply return the failed status from earlier.

In the `then` branch, we know that the USB transaction was successfully started. But the types tell us that we cannot return an URB that is "in progress." Therefore we use the function `usb_wait_while_active` to ensure that the URB is no longer active when it returns. Then we flush the transfer status using `urb_transfer_completed`, and deallocate the TDs using `urb_detach_and_free`. This version of the function does not return `void`, but rather it returns a status value that it obtains by examining the TDs.

With all this done, we can safely return from the anonymous function and allow `usb_with_urb` to complete the job of cleaning up the URB.

4.1.2 The provided library interface

Allocation and release

The `ehci` module identifies and configures all USB devices that it finds. Then it makes them available to other programs via an IPC interface that is mediated by the interface found in Listing 4.3. USB devices are indexed starting from 0 and applications can obtain a handle by requesting it. Since `usb_device_vt` is a viewtype, the type-checker ensures that it is later released. The interface is not sophisticated but suffices for the purpose of experiments.

```
absvtype urb_vt (i: int, nTDs: int, active: bool)

vtypedef urb0 = [i: int | i >= ~1] urb_vt (i, 0, false)

fun usb_device_alloc_urb
    {i, endpt, maxpkt: nat | (endpt < 16 && 7 < maxpkt && maxpkt < 2048) ||
                             (endpt == 0 && maxpkt == 0)} (
  ! usb_device_vt (i), int (endpt), int (maxpkt), & urb0? >> urb_vt (i', 0, false)
): #[s, i': int | (s != 0 || i' == i) && (s == 0 || i' == ~1)] status (s)

fun usb_device_release_urb {i: nat} (
  ! usb_device_vt (i), urb_vt (i, 0, false)
): void

prfun usb_device_release_null_urb (urb_vt (~1, 0, false)): void
```

Listing 4.4: URB allocation and release

A USB Request Block (URB) is the entity through which USB transfers are mediated. URBs are allocated within each USB device. Behind the scenes, URBs are actually EHCI Queue Head (QH) data structures, and the `ehci` module has preconfigured all of the possible QHs into the circularly-linked list that forms the Asynchronous schedule. An example of this is shown in Figure 4.2. The interface specified in Listing 4.4 allows application programs to retrieve a handle to a preconfigured URB that must eventually be released, thanks to the ATS type system. URBs are allocated with an endpoint and a maximum packet size specified. The USB specification puts some hard limits on those values, so that has been encoded into ATS. I have created a special case for endpoint 0, the Control endpoint, where by specifying maximum packet size 0, it will automatically look up the proper maximum packet size for endpoint 0.

Since it is possible for URB allocation to fail, the return types reflect the possibility that the returned URB is invalid, as given by a negative index. The alias `urb0` is a shortcut for an URB that could be invalid. The function `usb_device_alloc_urb` returns a status value to tell us if allocation succeeded; if OK then it stores the URB object into the "out parameter" that is passed by reference. The ATS type system forces you to check if the returned status is OK in order to prove that the URB has a valid index.

```
fun usb_with_urb
    {i, endpt, maxpkt: nat | (endpt < 16 && 7 < maxpkt && maxpkt < 2048) ||
    (endpt == 0 && maxpkt == 0)} (
  ! usb_device_vt i, int endpt, int maxpkt,
  f: &((! usb_device_vt i, ! urb_vt (i, 0, false)) -<clo1> [s: int] status (s))
): [s: int] status (s);
```

Listing 4.5: `usb_with_urb`

```
absvtype usb_mem_vt (l: addr, n: int)

fun usb_buf_alloc {n: nat | n < USB_BUF_SIZE} (int n): [l: agez] usb_mem_vt (l, n)

fun usb_buf_release {l: agz} {n: int} (usb_mem_vt (l, n)): void

prfun usb_buf_release_null {a: t@ype} {n: int} (usb_mem_vt (null, n)): void
```

Listing 4.6: USB memory area allocation and release

This is encoded by the logical statement `(s != 0 || i' == i) && (s == 0 || i' == ~1)` where OK is considered to be a status value of 0. If the URB turns out to be invalid, a trivial proof-function is provided to discard the linear object, which is named `usb_device_release_null_urb`. Proof-functions are erased by the compiler, so this is just a measure that exists to tie up loose ends. For valid URBs, the real function `usb_device_release_urb` takes care of cleaning up.

Allocation and release of an URB is a rote procedure, so I have also provided a helper higher-order function, shown in Listing 4.5, that can be applied to a function with a stack-allocated closure. An example use and detailed explanation is found in Section 4.1.1.

In order to send or receive data via USB, a buffer must be prepared in a special memory region that is designated for memory-mapped I/O. The details are handled by an interface shown in Listing 4.6. Again, it is possible for allocation to fail, so `usb_mem_vt` is indexed by an address value that can be zero. This situation occurs so often in ATS that there is a set of aliases for convenience:

- `addr` is the general, unrestricted sort that models pointer addresses.

```
fun usb_buf_set_uint_at {l: agz} {i, n: nat | 4 * i <= n} (
  ! usb_mem_vt (l, n), int (i), uint
): void

fun usb_buf_get_uint_at {l: agz} {i, n: nat | 4 * i <= n} (
  ! usb_mem_vt (l, n), int (i)
): uint

fun{a:t@ype} usb_buf_copy_from_array {n, m: nat} {src, dst: agz} (
  ! (@[a][m]) @ src |
  ! usb_mem_vt (dst, n), ptr (src), int (m)
): void

fun{a:t@ype} usb_buf_copy_into_array {n, m: nat} {src, dst: agz} (
  ! (@[a][m]) @ dst |
  ptr (dst), ! usb_mem_vt (src, n), int (m)
): void

fun usb_buf_takeout {a: t@ype} {l: agz} {n: nat | sizeof (a) <= n} (
  usb_mem_vt (l, n)
): (a? @ l, a @ l -<lin,prf> usb_mem_vt (l, n) | ptr (l))
```

Listing 4.7: USB memory area access

- `agez` means "address greater than or equal to zero."
- `agz` means "address greater than zero."

Therefore, by returning `[l: agez] usb_mem_vt (l, n)`, the function `usb_buf_alloc` is telling us that the return value might be NULL. But most functions that want to handle `usb_mem_vt` will require an `agz`, that is, an "address greater than zero," including `usb_buf_release`. To handle this situation in general, the ATS style is to overload the `ptrcast` function. This allows the programmer to compare the `usb_mem_vt` value against zero using an if-statement. If the value is zero, then it can be discharged using the proof-function `usb_buf_release_null`. And if it is greater than zero, the ATS type-checker is smart enough to understand the refined constraint.

USB device memory access

The memory regions used to transfer data to and from USB devices are described by a special viewtype named `usb_mem_vt`. However, ultimately, these are simply memory

```
datatype usb_RequestType_dir = HostToDevice | DeviceToHost
datatype usb_RequestType_type = Standard | Class | Vendor
datatype usb_RequestType_receipt = Device | Iface | Endpoint | Other
fun make_RequestType (
  usb_RequestType_dir, usb_RequestType_type, usb_RequestType_receipt
): usb_RequestType_t

datatype usb_Request =
  GetStatus | ClearFeature | _Invalid_bRequest1 | SetFeature |
  _Invalid_bRequest2 | SetAddress | GetDescriptor | SetDescriptor |
  GetConfiguration | SetConfiguration | GetInterface | SetInterface |
  SynchFrame
fun make_Request (usb_Request): usb_Request_t
```

Listing 4.8: USB request types and requests

regions that will need to be accessed by ordinary program code at some point. Several functions are available for that purpose, shown in Listing 4.7. These were developed as needed for various experiments, as I found a use for them. `usb_buf_set/get_uint_at` are basic buffer indexing operations that operate on a single machine word, a `uint`. This is useful for tasks such as quickly accessing the header of a packet stored in the USB memory space. For larger operations, there are memory copy functions for transfering data between an ATS array and the USB memory space. Finally, most generally, `usb_buf_takeout` allows a programmer to temporarily cast the USB memory space as an ordinary pointer with any view desired; as long as the size requested fits within the space. This function generates a linear proof-function that serves as an obligation to convert the pointer back into a `usb_mem_vt` so that the ordinary pointer cannot "escape" and be used in other contexts. The cast target type `a?` is considered to be an uninitialized location in ATS, with the question mark modifier, and is protected against uninitialized read by the type-checker.

Transfers

The USB allows devices to communicate with the host using what is known as a "control" transfer; the purpose being to help configure the hardware for further use.

The format of these transfers is well-defined within the USB specification. Without getting into the details of packet types, there are essentially 6 parts to a control transfer: request type, request, value, index, length, and data. The request type and request values have been defined in ATS as shown in Listing 4.8. The request type describes the direction of data transfer, the type of request, and the recipient of the request. The request itself can be either one of the standard requests, or something defined in a class specification or by a vendor. If it is a standard request then it appears in Listing 4.8. Note that the use of `datatype` in ATS with these 0-parameter constructors is efficient: there is no memory allocation, instead, they are simply enumerated starting from 0. The function `make_RequestType` does the bit-manipulation necessary to form a single value from the three parameters.

Both `wValue` and `wIndex` are general-use parameters that depend upon the precise request being fulfilled for their meaning. All of the standard requests specify what they expect these values to be, and so should all vendor and class requests.

Listing 4.9 shows the function signature for `urb_begin_control_read`, which is one of the functions available for initiating control transfers. There is also a symmetric function for control write, and also one for what is called "control nodata" for when there is no data being transfered. The control write function has the same signature

```
fun urb_begin_control_read {i, n, len: nat | len <= n} {l: agz} (
  ! urb_vt (i, 0, false) >> urb_vt (i, nTDs, active),
  usb_RequestType_t,
  usb_Request_t,
  int, // wValue
  int, // wIndex
  int (len), // wLength
  ! usb_mem_vt (l, n) // data
): #[s: int]
   #[nTDs: int | (s == 0 || nTDs == 0) && (s <> 0 || nTDs > 0)]
   #[active: bool | (s == 0) == active]
   (usb_transfer_status (i) | status (s))
```

Listing 4.9: USB control read

```
fun urb_begin_bulk_read {i, n, len: nat | len <= n} {l: agz} (
  ! urb_vt (i, 0, false) >> urb_vt (i, nTDs, active),
  int (len), // number of elements
  ! usb_mem_vt (l, n) // data
): #[s: int]
   #[nTDs: int | (s == 0 || nTDs == 0) && (s <> 0 || nTDs > 0)]
   #[active: bool | (s == 0) == active]
   (usb_transfer_status (i) | status (s))
```

Listing 4.10: USB bulk read

and the only difference for "nodata" is that `wLength` and `data` are no longer needed. Those two functions have been omitted for space's sake.

Examining `urb_begin_control_read` more closely, you can see that the first parameter is a `urb_vt` that is expected to evolve upon return. Initially, it must provably signify a URB that has zero TDs attached and that is not active. But afterwards, the number of TDs attached and active state will depend upon whether the function was able to successfully return or not. Therefore, in the return type, the indices have been defined with the following logic in mind: if and only if `s` is OK, then `nTDs` is greater than zero. And since `active` is a boolean value, it can simply be assigned the result of testing whether `s` is OK. The net effect of providing this return type is that the programmer is forced to check the returned status of the function in order to write sensible code that will pass the type-checker.

The remainder of the parameters are as described by the USB specification, except that the data parameter must be provided as the special `usb_mem_vt` viewtype that was discussed earlier. Note that the ATS type-checker enforces the provision `len <= n` that the `wLength` parameter must be provably no larger than the amount of USB memory allocated. Finally, on the proof-world side, the function also returns a view named `usb_transfer_status` that also serves as an obligation to check and clean-up after the transfer. Such functions will be discussed shortly.

Before moving on, I would like to describe the function used for initiating the USB

```
fun urb_transfer_chain_active {i, nTDs: nat | nTDs > 0} {active: bool} (
  ! usb_transfer_status (i) |
  ! urb_vt (i, nTDs, active) >> urb_vt (i, nTDs, active')
): #[s: int] #[active': bool | (s == 0) == active'] status (s)

fun urb_transfer_result_status {i, nTDs: nat | nTDs > 0} (
  ! urb_vt (i, nTDs, false)
): [s: int] status (s)

fun urb_wait_if_active {i, nTDs: nat} {active: bool} (
  xfer_v: ! usb_transfer_status (i) |
  usbd: ! urb_vt (i, nTDs, active) >> urb_vt (i, nTDs, false)
): void
```

Listing 4.11: Checking result status

"bulk" transfer process. Bulk transfers are the USB's mechanism for transfering large amounts of general-purpose data on an available-bandwidth basis. They are actually rather similar to control transfers except that much less is required to be specified. All that a bulk transfer needs is a URB, a length, and the data buffer. Listing 4.10 shows the signature of the bulk read function, which is symmetrical to the bulk write function. There is no "bulk nodata" function, since that would not make sense. As you can see, it is very similar to the control read function, but with fewer parameters, so I will not repeat the discussion again.

Checking result status

Upon initiation of the transfer, the USB host controller will asynchronously send and receive packets at the next available opportunity. The outcome of the transfer will be reflected in the special data structures that are used by the driver to communicate with the host controller. The precise details of those data structures are kept abstract by this interface. Instead, the programmer has several functions available to query the status of the transaction: `urb_transfer_chain_active` in Listing 4.11 returns `OK` if the host controller has not completed the transfer. The return type of the function specifies that the value of `active'` has the same boolean value as comparing `s == OK`.

```
fun urb_transfer_completed {i, nTDs: nat} (
  usb_transfer_status (i) |
  ! urb_vt (i, nTDs, false)
): void

fun urb_transfer_abort {i, nTDs: nat} {active: bool} (
  usb_transfer_status (i) |
  ! urb_vt (i, nTDs, active) >> urb_vt (i, nTDs, false)
): void

fun urb_detach {i, nTDs: nat} (
  ! urb_vt (i, nTDs, false) >> urb_vt (i, 0, false)
): [l: agz] ehci_td_ptr (l)

fun urb_detach_and_free {i, nTDs: nat | nTDs > 0} (
  ! urb_vt (i, nTDs, false) >> urb_vt (i, 0, false)
): [s: int] status s

fun usb_device_detach_and_release_urb {i, nTDs: nat | nTDs > 0} (
  ! usb_device_vt (i), urb_vt (i, nTDs, false)
): [s: int] status s
```

Listing 4.12: Clean up

This allows a programmer to resolve whether or not the transfer is still active by using an ordinary `if`-statement, and the indexed type `urb_vt` will be adjusted appropriately depending upon which branch is considered.

Once the program reaches a point where `active` index is proven to be false, then it is possible to query the final result status of the transfer. If it was successful, then `urb_transfer_result_status` will return `OK`, or otherwise it will return an error status such as `EDATA` or `EINCOMPLETE`. Now, it is possible for the program to go and do other things while waiting for the transfer to become inactive, but a convenience function is also provided that busy-waits until the transfer is complete: `usb_wait_if_active`. The behavior is described by its type: it will only return once it has proven that the URB is no longer active.

Clean-up

Listing 4.12 shows signatures of some functions that assist with cleaning up after a USB transaction. The first function, `usb_transfer_completed`, can only be used on an inactive URB and it consumes the `usb_transfer_status` view that was produced by the initiation function. The second function, `usb_transfer_abort`, can be used on an URB in any state of activity, and is intended to be used when the programmer wishes to cancel a transfer. Once a URB is inactive, then it can have its TD chain safely removed. The TD chain is a linked list of host controller data structures used to communicate with hardware. It is important that the URB be inactive before tinkering with the TD chain or else hardware may become confused. The ATS type system prevents the use of these functions until the URB is provably inactive.

The most basic function is `urb_detach` that unlinks the TD chain from the URB and then returns it. TD chains are pieces of device memory, and therefore represent a resource that must be handled by the programmer correctly. I have not discussed TD chains yet because they are a low-level detail that is mostly unnecessary to handle when using the high-level interface. But if the programmer wishes to obtain a handle on the TD chain, then it must eventually be freed with a call to `ehci_td_chain_free`.

More likely, the programmer will want to simply combine those two steps into one that avoids the need to handle the TD chain explicitly, by using `urb_detach_and_free`. This function detaches the TD chain, then frees the memory, but not before obtaining the result status of the transfer – which it then returns. Altogether a convenient function. There is also a version named `urb_detach_and_free_` that does not return the status.

Finally, there is a function named `usb_device_detach_and_release_urb` that not only detaches and releases the TD chain, it also releases the URB. This is useful for functions that execute a single USB transfer with an URB and quickly want to clean up and return a status.

```
fun usb_set_configuration {i: nat} (! usb_device_vt (i), int): [s: int] status (s)

fun usb_get_configuration {i: nat} (
  ! usb_device_vt (i), & uint8? >> uint8
): [s: int] status (s)

fun usb_get_endpoint_status {i: nat} (
  ! usb_device_vt (i), int, & uint? >> uint
): [s: int] status (s)

fun usb_clear_endpoint {i: nat} (! usb_device_vt (i), int): [s: int] status (s)
```

Listing 4.13: Helper functions

Helper functions

Using these USB transfer tools, several basic helper functions shown in Listing 4.13 have been provided that execute some of the standard device requests allowed for by the USB standard. One of the most important is `usb_set_configuration`: it selects the configuration of the device based upon the provided configuration value (a simple integer). Just about all USB devices require that this configuration step take place before operation may proceed. It has a counterpart named `usb_get_configuration` that shows how such a function might choose to return a result value. In this case, the second parameter is by reference, and it specifies that some piece of uninitialized 1-byte length piece of memory be provided to write the returned value. The actual return value is a status that specifies whether or not the function succeeded. Both of the other functions behave similarly and perform their respective duties as defined by the USB specification.

4.1.3 Performance

Figure 4.3 shows the results of an experiment conducted using two separate USB Ethernet network interface cards (NIC). NIC1 is from the SMSC 9500-series, and NIC2 is based on the ASIX AX88772, both commonly used by USB Ethernet adapters. The NICs were both hooked up to a third machine, a quad-core Xeon E5506 running

	CPU0		CPU1	
	UART	NIC1	EHCI	NIC2
Capacity	1	C	1	18
Period	20	20	20	20

Table 4.1: Basic scheduling setup for the experiments

Scientific Linux 6, using an Ethernet switch (but this was changed later, as explained below). On the PandaBoard, four processes were loaded and arranged into a static priority rate-monotonic schedule across the two processors, as laid out by Table 4.1. The entry marked with an C, the capacity of NIC1, was varied over the course of the experiments. Capacity and Period units are given in multiples of 1/16384 of a second, approximately 61 μseconds each.

Both of the NIC processes are continually running a copy of the μIP (Dunkels, 2013) TCP/IP stack each, independently responding to network activity by polling the interfaces. They are both using the provided USB library and interface for ATS in order to operate and manage their own memory-mapped I/O interaction with the EHCI host controller hardware itself. Both drivers are able to perform their duties without invoking any kernel functionality, and with only one exception they are operating entirely in their own space. That main notable exception is the way the Queue Heads must be managed in the shared device space, as described by the EHCI specification. Access to the Queue Head data structures is mediated through the ATS programming interface for safety purposes. The arrangement is described by Figure 4.1, which shows how the `ehci` module uses the IPC framework to communicate status information, as well as set up the USB memory-mapped device I/O space for both NIC modules to use independently.

For the purpose of this experiment, one NIC at a time was bombarded with ICMP packets. In Figure 4.3, we can see that reducing the capacity of the NIC1 process from 19 down to 2 has a drastic effect on its ability to respond to ICMP ping request packets. I have included two data-sets on the chart, in order to elaborate on

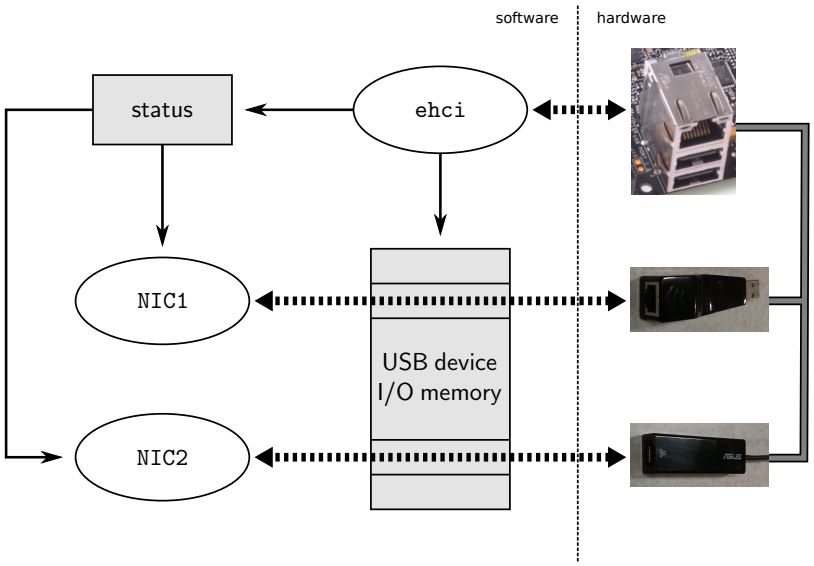

Figure 4.1: The relationships between modules and hardware

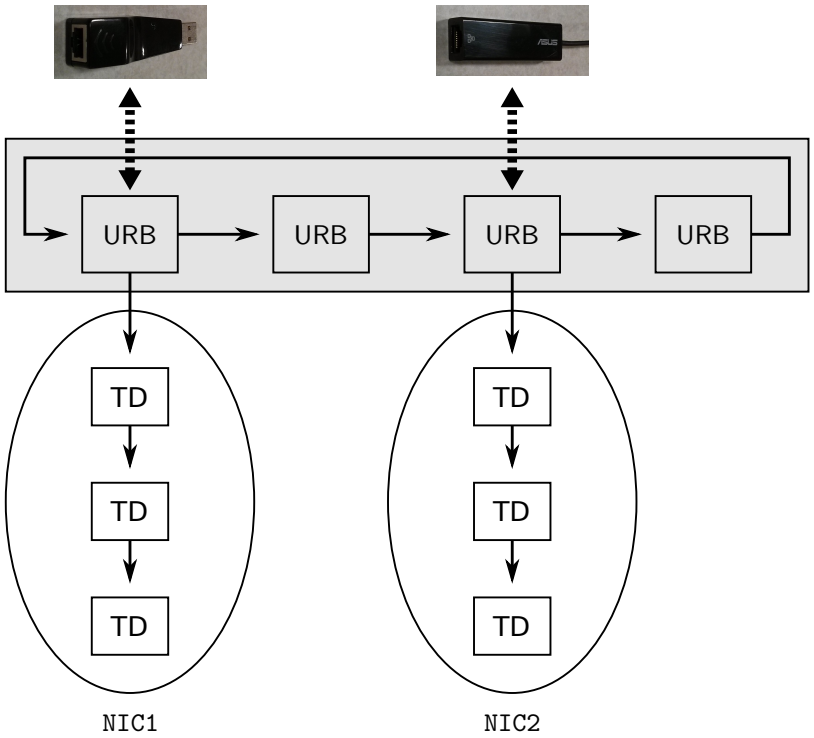

Figure 4.2: URB linked list example

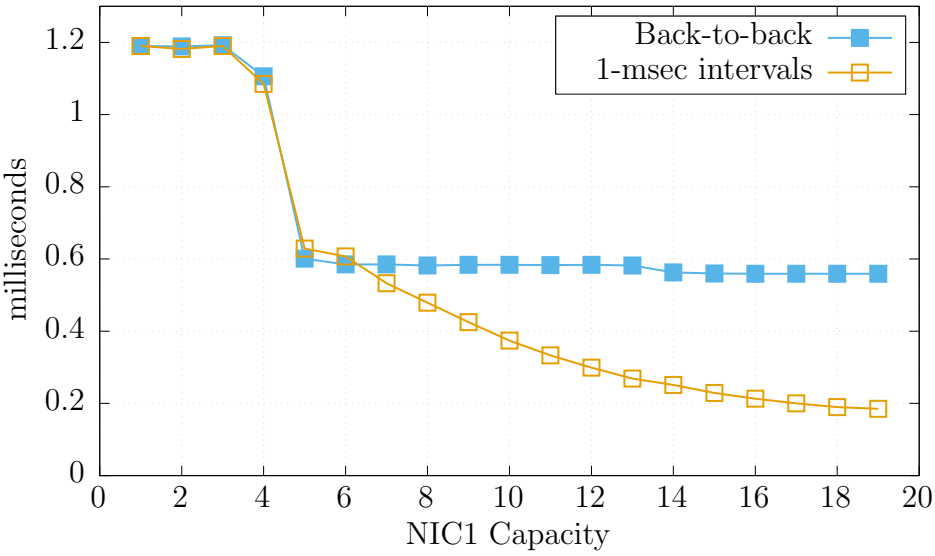

Figure 4.3: Effect of NIC1 capacity on ping roundtrip time using Ethernet switch

a commonly observed effect. The experiments were conducted under two scenarios: one where ICMP ping packets were released as soon as the previous response was received ("Back-to-back"), and one where ICMP ping packets were released at precisely 1 millisecond intervals ("1-msec intervals"). As shown on the chart, these set of experiments wound up demonstrating that there is a difference between response time and interpacket gap.

Since response times were normally well under a millisecond, when a single packet was sent every millisecond, the USB NIC was able to respond within about 200 μsec each time. However, when packets were sent out as fast as possible – back-to-back – the response times rose to around 550 μsec after the first packet. You can see this effect by observing how the "Back-to-back" trend line flattens out at just under 0.6 msec. The same effect occurs with both NICs and also under Linux. Then I tested it using an old-fashioned Ethernet hub as well as a direct connection and found that the discrepancy disappeared. The results of comparing the Ethernet switch to a direct connection are shown in Figure 4.4. I am forced to conclude that it is

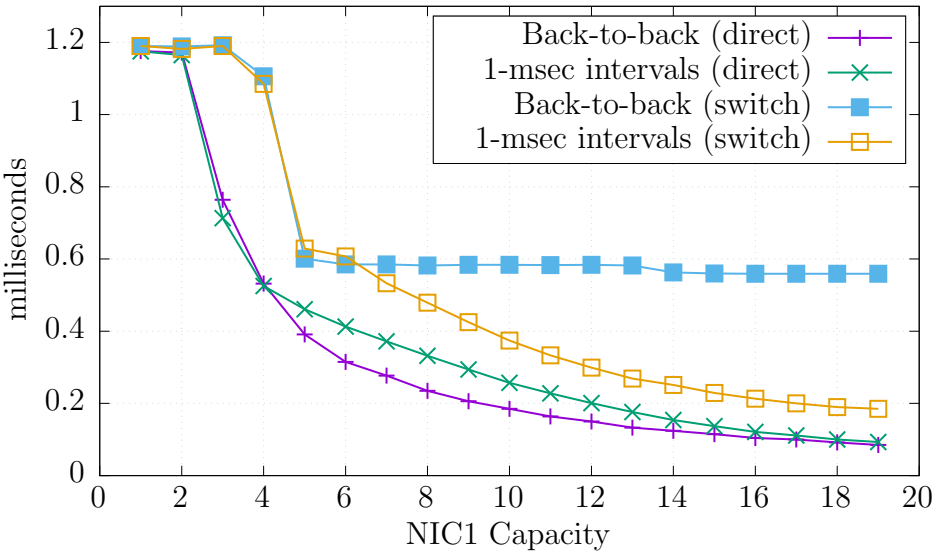

Figure 4.4: Effect from type of network connection on ping roundtrip time

the Ethernet switch, a Netgear GS105, that is responsible for forcing the apparent 550 μsec minimum inter-packet gap during the "back-to-back" test. Therefore, to eliminate as much interference as possible, further testing was done without the use of the Ethernet switch. Instead, a direct connection was formed by plugging a cross-over cable into each NIC.

After sorting that out, to give a comparison, I booted the PandaBoard using a vendor-supplied image of Angstrom 2010.4-test-20100416 running under Linux kernel version 3.0.4 (SMP) for ARM. And for an unfair comparison, I also plugged one of the USB Ethernet devices into a four-core Xeon® E5506 (2.13 GHz) running Scientific Linux 6. Also, even more unfairly, I ran an additional test against the PCI-based NIC of the same machine. These results are plotted in Figure 4.5, along with the Terrier USB NIC performance under different levels of capacity. As you can see, the most responsive network adapter is the PCI NIC under SciLinux6, unsurprising given the quality of the interconnect and the machine. All of the USB NICs operated at a disadvantage to the PCI card. The performance of the Terrier driver depends

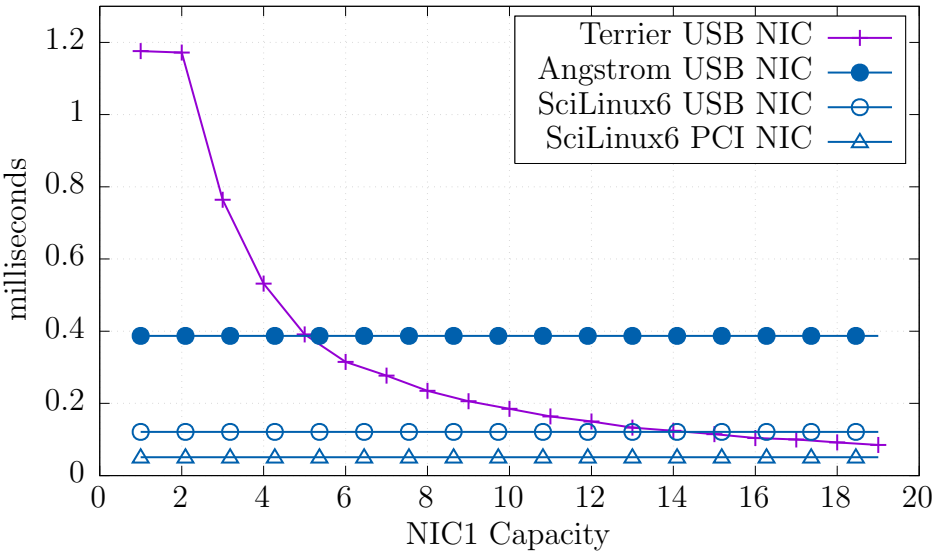

Figure 4.5: Comparison of ping roundtrip time to various Linux scenarios

upon the amount of capacity given. When $C > 14$, Terrier manages to respond more quickly than SciLinux running on the quad-core Xeon. And as long as $C > 5$, Terrier is more responsive than Angstrom Linux running on the same platform.

Linux is using a more conventional USB stack with interrupt handlers that check the status of the USB transaction upon IRQ. Terrier is running processes that are devoted to periodically polling the USB transaction data structures according to the static priority schedule that is configured ahead of time. With less overhead going around, Terrier is capable of responding more quickly to packets when the NIC process is given enough capacity.

Returning to Figure 4.3, as capacity is lowered, we should expect and do see that some degradation of performance occurs: going from 90% CPU utilization down to 50% CPU utilization means that the NIC1 process can only respond immediately to ping requests about half of the time. That fact is made evident in the distribution of ping response times as displayed in the summary of results, shown on Figure 4.6. The more capacity given to the driver, the closer the minimum is to the maximum;

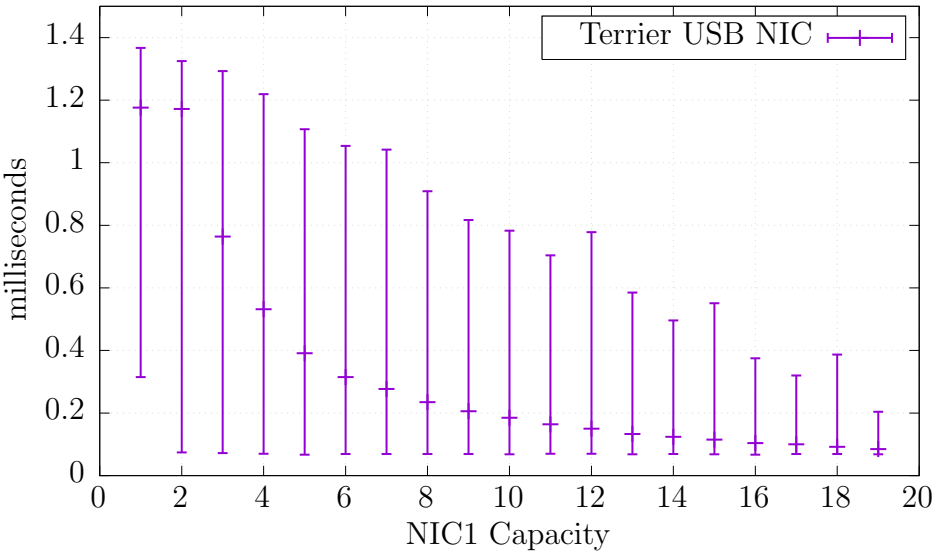

Figure 4.6: Distribution of ping response times (min/avg/max)

the less capacity given, the larger the difference and the higher the average.

The minimum roundtrip time reflects a lucky ping request that arrived while the NIC1 process was running and ready to respond, whereas the maximum roundtrip time is the result of a packet that arrived just as the process ran out of its allotted capacity for that period. This effect can be seen further in Figure 4.7, where the period of NIC1 is extended while maintaining a 90% CPU utilization through corresponding capacity increases. The longer the period, the larger the worst-case response time.

Although fast ping roundtrip time is not an explicit goal of the Terrier project, I am pleased to see that the inside-out approach of Terrier's USB interface is vindicated by this strongly responsive performance in experiments; with quality able to be smoothly controlled through adjustment of the scheduling parameters. Terrier is able to respond in under 100 μsec when utilization is near 100%, and can still respond more quickly than Linux even when utilization is reduced almost to 25%. This is despite the fact that the Terrier drivers are relatively crude and unrefined compared to the years of work behind Linux USB drivers and TCP/IP stack. The ability of the NIC1 program,

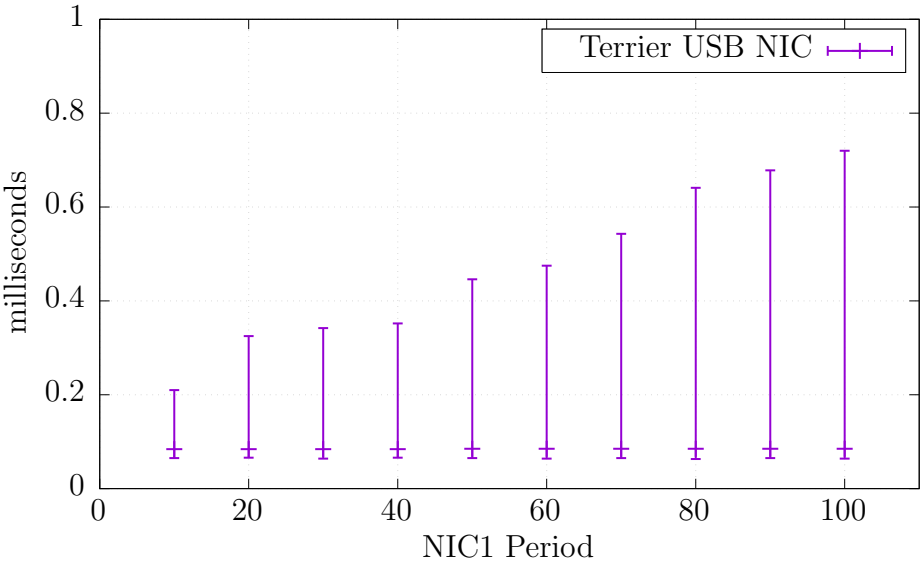

Figure 4.7: Distribution of ping response times in experiment that varies period of NIC1 while maintaining 90% utilization for capacity

running on the PandaBoard, to manage its own USB data structures without kernel intervention, or other context switching, allows it to match responsiveness levels found on the much more powerful PC hardware.

4.2 Physical memory manager

In Section 2.3.1 a brief overview of the physical memory manager was given. This section will continue the discussion. The physical memory manager is largely implemented in a functional style in ATS, providing a useful example of how some prosaic, run-of-the-mill datatypes can be deployed for typical, straightforward data structure manipulation.

Listing 4.14 shows the type signature and implementation of the function named `physical_alloc_pages` found in the file `mem/physical.dats`. Physical memory is tracked by a set of bitmaps where each bit corresponds to a physical frame in available memory: 4096 kilobytes each. The allocation function uses a linear search of these bitmaps in order to find a consecutive string of available, suitably-aligned, frames. The `search`

```
fun physical_alloc_pages (
    n: int,
    align: int,
    addr: & physaddr? >> physaddr_t p
  ): #[s: int] #[p: addr | s == OK <==> p > null] status s

implement physical_alloc_pages(n, align, addr) = let
  val (n', saved) = search (n, align, n, @(0, 0, 0), @(~1, 0, 0))
  val (saved_m, saved_byt, saved_bit) = saved
in
  if saved_m = ~1 || n' <> 0 then begin
    addr := nullphysaddr; ENOSPACE
  end else begin
    mark (n, saved);
    addr := bitmap_compute_start_physaddr (saved_m, saved_byt, saved_bit); OK
  end
end
```

Listing 4.14: `physical_alloc_pages`

function is shown in Listing 4.15. The main complication of the search function comes from the fact that it must find `n` available frames in a row, and therefore must restart the count whenever an allocated frame interferes with the search.

The current position of the search is stored as a flat tuple of `int`s, as described by the type named `searchstate`. Since ATS translates these flat tuples into C structs, these usages of the `searchstate` type, and even the return type `@(int, searchstate)`, are all compiled to efficient binary code making use of registers and the stack to handle dataflow. A `searchstate` encapsulates the three numbers that describe a location in the bitmaps: bitmap index, byte index, and bit offset. The function `inc_searchstate` returns a new state that has been incremented to the next consecutive state. And using this function, `search` uses tail-recursion to loop through the bitmaps until it finds the consecutive string of available frames, or reaches the end.

Finally, Listing 4.16 shows a simple tail-recursive function that loops from a starting point over an identified range of consecutive frames, and marks each one. Both `search` and `mark` use basic functional programming techniques such as tail-recursion and pattern matching to implement the algorithm. Concurrency issues are avoided

by having the physical memory manager only be operated on a single processor; within the framework of Terrier, thus far, that has not been a problem because most allocation takes place during the boot process.

```
typedef searchstate = @(int, int, int) // (m, byt, bit)

fun inc_searchstate((m, byt, bit): searchstate): searchstate =
  if bit + 1 < 8 then (m, byt, bit + 1)
  else if byt + 1 < bitmap_num_bytes m then (m, byt + 1, 0)
  else (m + 1, 0, 0)

fun search (
    n: int, align: int, count: int , curr: searchstate, saved: searchstate
  ): @(int, searchstate) =
let
  val (m, byt, bit) = curr
  val curr' = inc_searchstate curr
in
  if m = num_bitmaps then
    // reached end of bitmap sequence
    (count, saved)
  else if count = 0 then
    // found n unallocated pages - terminate search
    (0, saved)
  else if bitmap_tst (m, byt, bit) then
    // found an allocated page - reset count
    search (n, align, n, curr', @(~1, 0, 0))
  else if m <> curr'.0 then
    // cannot have ranges spanning bitmaps - reset count
    search (n, align, n, curr', @(~1, 0, 0))
  else if count = n && is_aligned (bit, align) then
    // found an unallocated, aligned page - start span
    search (n, align, count - 1, curr', @(m, byt, bit))
  else if count = n then
    // found an unallocated, unaligned page - skip it
    search (n, align, count, curr', @(~1, 0, 0))
  else
    // found an unallocated page mid-span - keep counting down
    search (n, align, count - 1, curr', saved)
end
```

Listing 4.15: `search` and search states

```
fun mark (count: int, curr: searchstate): void = let
  val (m: int, byt: int, bit: int) = curr
  val curr' = inc_searchstate curr
in
  if count > 0 then begin
    bitmap_set (m, byt, bit);
    mark (count - 1, curr')
  end else ()
end
```

Listing 4.16: `mark`

Chapter 5

Debugging with Types and Logic

The Terrier operating system is written partially in the ATS programming language because I need to be able to write very low-level code, but at the same time, I would like to enjoy the advantage of a very strong type system. ATS offers what are known as dependent and linear types, which can be used to create very precise specifications of intended program behavior that are then checked statically prior to compilation. The output of ATS is C code that may be compiled and linked with ordinary C compilers such as GCC, and it does not require any special run-time support.

In the Terrier OS, all of the interprocess communication mechanisms are written in ATS and must be invoked through an ATS-based API. This allows me to do a great deal of safety checking prior to compilation. The fixed-slot mechanism is no different, and it has been developed under the same requirements. Furthermore, the use of ATS types during the process of designing the algorithm resulted in the finding of a subtle concurrency bug that has since been corrected in the final design. That bug will be discussed in Section 5.1.1.

5.1 The fixed-slot mechanism

The next few sections will describe how I used ATS to help me design and specify the algorithms for the fixed-slot mechanism, assuming no reader knowledge of ATS. The code presented is a simplified form of the actual code, but the salient points will all be covered.

5.1.1 Static types for the fixed-slot mechanism

Before a single line of assembly code or actual working code was written, I began to sketch out the overall skeleton of the reader and the writer using a series of function prototypes, along with abstract data types. The first step was establishing a name for the abstract data type that would represent a fixed-slot mechanism: `absviewtype fixedslot`. The ATS declaration of `absviewtype` establishes a viewtype, also known as a linear abstract data type. Essentially, `fixedslot` is a type of data that will be treated as a resource at the type level. The way the program manipulates values of this type will be carefully watched by the compiler to ensure that they are used in a linear fashion: no aliasing allowed. I find linear types to be very helpful when managing anything that represents a system resource, such as blocks of memory, because it means that the type system prevents resource leaks and requires proper clean-up along all possible paths of execution.

The writer

I sketched out function prototypes for the atomic operation W1: in ATS, `fun pick_wi (! fixedslot): int`. This describes a function that accepts a single parameter of type `fixedslot` and returns an `int`. The importance of the `!` operator relates to the linear type system: normally, a value of a linear type is "consumed" when it is used, and thereafter becomes unavailable for use. If you want to create a linear sequence of operations on the same linear value, then you need to return or "reproduce" the value after each operation so that the next one can use it too. This gets to be tiresome to write out every time, so ATS provides a `!` notation that is syntactic sugar indicating that the function does *not* consume the linear value, leaving it for a future operation to consume instead. So, in essence, this function prototype for `pick_wi` simply states that the fixed-slot mechanism is at first consumed by this function, but then reproduced along with an integer value when it returns.

In a similar fashion, I can define all three steps of the writer function:

```
    absviewtype fixedslot
W1  fun pick_wi (!  fixedslot):  int
W2  fun{ty:  type} write_data (
    !  fixedslot, int, ty
    ):  void
W3  fun set_f (!  fixedslot, int):  void
```

The only additional feature introduced here is the use of ATS templates. By specifying `fun{ty:  type}` I can parameterize the `write_data` function over any type of data that the mechanism wants to write into memory, and that data would be passed as the third parameter to that function.

So far, I have done only the most minimal of specification. ATS is capable of far more, but I like to use a development process where I iteratively increase the strength of my specifications as I become more comfortable with the ideas behind my code. In this case, the next step would be to start trying to represent the shared state variables in an abstract way, statically, using what are called dependent types. The way I do that is by augmenting the abstract viewtype with static indices: `absviewtype fixedslot (p:  int, t:  int, f:  int)`. The indices `p`, `t`, `f` are not regular variables, they are variables at the type level. And the name `int` in this context does not mean a type of value, it means a type of a static term. These types are called "sorts" in ATS parlance, to distinguish them from types of values. Now, I am able to rewrite my function prototypes and make use of these indices to specify more precise behavior:

```
    absviewtype fixedslot (p:int, t:int, f:int)
W1  fun pick_wi {p,t,f:  nat} (
    !  fixedslot (p, t, f)
    ):  [i:  nat | i != p && i != t && i != f] int(i)
W2  fun{ty:  type} write_data
```

```
    {p,t,f,i:  nat | i != p && i != t && i != f} (
    !  fixedslot (p, t, f), int(i), ty
    ):  void
W3  fun set_f {p,t,f,f':  nat} (
    !  fixedslot (p, t, f), int(f')):  void
```

This requires some additional explanation: now that I have names for type indices, I must declare them. This is done through quantification. So for example, the syntax for universal quantification `{p,t,f:  nat}` reads as "for all `p`, `t`, and `f` that are natural numbers", where `nat` is an alias for `int`s greater than or equal to zero. And the syntax for existential quantification `[i:  nat | i != p && i != t && i != f]` reads as "there exists `i` that is a natural number such that `i` is not equal to `p`, `i` is not equal to `t`, and `i` is not equal to `f`." As you can see, ATS provides a expressive notation for specifying constraints on quantification in a manner that is reminiscent of mathematical set notation. In addition, you can see that `fixedslot` is not the only type that accepts indices. Many types can have indices. In this case, even the `int` type can have an index. The type `int(i)` denotes the type of integers that are exactly equal to `i`. So, if `i` is somehow determined to be equal to 1, then `int(i)` would mean that the only permissible value of that singleton type is the value of 1, therefore the type checker must be able to prove that is the case. None of these type annotations, indices or quantifiers is going to exist at run-time — they will all be erased by the compiler.

For the writer, I am mostly interested in showing that the value of `i` given to `write_data` is not going to conflict with any of the readers, nor is it going to overwrite the most recently written data. From the algorithm I know that the readers are supposed to obey an invariant that restricts them to operating only on the slots marked by p and t, as well as that the writer has marked the most recent data with f. Therefore, my precondition on `write_data` is simply that `i` is not equal to any of

those markers. The return value of `pick_wi` is purposefully intended to meet those same preconditions. Now, if my specification was actually complete, then I would have a very trivial proof, since these statements match exactly. But I am not finished with the specification: because of concurrency, I am making far too strong an assumption about the values of `p` and `t`. I do not know if `p` and `t` will retain the same exact values from step to step, because the readers may modify these values at any time. I only know that `f` will not be modified, except by the function `set_f` that only the writer invokes. So armed with this information, I need to modify my code a bit more:

```
    absviewtype fixedslot (p:int, t:int, f:int)
    viewtypedef postcon (p:int, t:int, f:int) = [
    p',t':  nat |
    ((p' == p && t' == t) || (p' == t && t' == t) ||
    (p' == t && t' == f) || (p' == f && t' == f))
    ] fixedslot (p', t', f)
W1  fun pick_wi {p,t,f:  nat} (
    !  fixedslot (p, t, f) >> postcon (p, t, f)
    ):  [i:  nat | i != p && i != t && i != f] int(i)
W2  fun{ty:  type} write_data
    {p,t,f,i:  nat | i != p && i != t && i != f} (
    !  fixedslot (p, t, f) >> postcon (p, t, f),
    int(i), ty
    ):  void
W3  fun set_f {p,t,f,f':  nat} (
    !  fixedslot (p, t, f) >> postcon (p, t, f'),
    int(f')
    ):  void
```

For convenience, I have defined an alias named `postcon` that uses existential quantification to specify a post-condition on every step. That post-condition states that there will exist two values of `p'` and `t'` such that one of four possible states can occur, by the definition of the algorithm, at every step. Those four states encode the legally allowed movements of the p and t markers, assuming that the f marker stays

constant.

The post-condition is then used in conjunction with a bit more ATS syntactic sugar. Earlier it was said that the `!` notation means that the function preserves a value of linear type. When type indices are involved, sometimes you want to do more than just preserve, you want the type indices to change after the function returns. Therefore, ATS includes a convenience syntax that allows you to specify that a linear, dependent type has not only been preserved, but it has returned in a different form. That syntax is constructed by combining the use of the `!` notation along with the `>>` notation as shown above. For example, in the function `set_f`, the notation `! fixedslot (p, t, f) >> postcon (p, t, f')` means that the linear value of type `fixedslot (p, t, f)` will become a linear value of the type aliased by `postcon (p, t, f')` when that function returns. In this case, it is useful for specifying that the value `int(f')`, supplied as the second parameter, is actually incorporated into the state of the fixed-slot mechanism.

At this point, I felt that I had specified the writer enough to move onto other parts of the code, before coming back to fill in the actual working code. The ATS type-checker is able to verify that this type of `write_data` is satisfied by the type of value returned by `pick_wi`. Even though it is not quite as trivial as before, the constraints are simple enough for an automated solver, included with ATS, to verify.

The reader

After putting together the writer, a similar evolution of specifications was worked out for the reader.

```
R1 fun pick_ri {p,t,f: nat} (
   ! fixedslot (p, t, f) >> fixedslot (p', t', f')
   ): [p',t',f',i: nat | ((p == t) ==> (t' == f')) && i == t'] int(i)
```

The function `pick_ri` returns an integer and morphs my `fixedslot` viewtype in a similar way to that described in the previous section. Except in this case, I can no longer assume that `f` is untouched. Now I must assume that any of `p`, `t`, or `f` can be changed behind my back. From analysis of the algorithm, I do know that the new value of `t'` could be equal to `f'` if in the pre-condition it was true that `p == t`. And the return value from this step will be equal to `t'`, either way.

The following step introduces a new abstract linear proposition, also known in in ATS as an abstract view. A linear proposition must be managed as a resource at the type level, much like the viewtype `fixedslot`. But because it is merely a proposition, it does not exist at the program level, only in the type-checker. It will be erased along with the rest of the types, by the ATS compiler. Therefore, it is useful for representing some kind of property, statically, without interfering with run-time performance.

```
    absview reading_view (i:  int)
R2  fun incr_rcount {p,t,f,i:  nat} (
    !  fixedslot (p, t, f) >> fixedslot (p', t', f'),
    int(i)
    ):  [p',t',f':  nat | (p' == p && t' == t) ||
    (p == t && p' == p && t' == f')]
    (reading_view(i) | void)
R4  fun{ty:  type} read_data
    {p,t,f,i:  nat | i == p || i == t} (
    !  reading_view(i) |
    !  fixedslot (p, t, f) >> fixedslot (p', t', f')
    ):  [p',t',f':  nat] ty
```

In the case of R2, the `reading_view` is introduced to be a kind of reminder that I am in the process of reading, I have incremented a number in memory, and that operation must be cleaned up after the read is complete. The ATS type-checker will ensure that I do not forget to decrement the number after the read is complete.

This basic usage of views is quite common in ATS programs because it is simple and yet effective: it stops resource leaks. Since `reading_view` does not represent a real program value, it must be returned on the left side of a `|`, as shown by the syntax `(reading_view(i) | void)`. This is the so-called "proof" side of function arguments, and it consists of parameters that are completely erased by the compiler.

More importantly for the proof of my algorithm, I have a post-condition specified on the indices `p', t', f'`. This post-condition establishes that there can be only two possible alternatives for the shared state variables after this step returns: either there has been no change to the variables, or `t'` has now become equal to `f'`. It was in the process of writing this post-condition that I realized that the algorithm was incomplete as originally formulated.

The safety counter The shared state variable S was not part of the original formulation of the fixed-slot ACM. Prior to introducing S, the formulation of the post-condition for R2 had to be specified with more than two branches in the disjunction, in addition to what is shown above. After all, it was possible in theory for interleaving to produce several outcomes. In particular, it was possible for other reader threads to come along and move both the p and t markers around.

However, I knew for certain that I wanted to prove that my function `read_data` would be given an integer parameter `i` such that `i == p` or `i == t`. That's because one of the fundamental principles of the fixed-slot ACM is that the readers can operate only on one of two slots: the one marked by p or the one marked by t.

But this pre-condition of `read_data` simply was not provable when the post-condition of R2 could involve changes to `p`. After struggling with it for a while, I realized that in the original formulation of the reader algorithm there was a counter-example to the desired pre-condition. Namely, it was conceivable for a sequence of operations to interleave in such a fashion that the current r_i would be left behind while

Suppose $p = 2, t = 0, f = 1$ and $rcount = \{0, 0, 0\}$.

R_1 : $r_i \leftarrow t$

Now R_1's version of $r_i = 0$

R_2 : `if` $rcount\,[p] = 0$ `then` $p \leftarrow t$

Now $p = 0$.

R_2 : `if` $p = t$ `then` $t \leftarrow f$

Now $t = 1$.

R_3 : `if` $rcount\,[p] = 0$ `then` $p \leftarrow t$

Now $p = 1$.

R_1 : The invariant fails: $r_i = 0$ while $p = t = 1$.

Figure 5.1: What happens without the safety counter S.

the other readers pushed both the p and t markers to other slots. Then, once the current reader resumed, its value of r_i would violate the pre-condition of `read_data`. An example sequence with reader threads R_1, R_2, and R_3 is shown in Figure 5.1.

With this example in mind, I determined that the problem was due to the fact that the marker p could be moved around after a value of r_i had been chosen but before $rcount\,[r_i]$ had been incremented. I then decided to create the safety counter S with the purpose of preventing this kind of interleaving from taking place. Going back to Figure 3.16 you should now be able to see that the safety counter prevents the modification of p while any reader thread is working on steps R1, R2, and has not yet completed R3. Therefore, returning to the post-condition of R2, this fact simplifies the disjunction down to the two branches you do see in the above code: `(p' == p && t' == t) || (p == t && p' == p && t' == f')`. With this post-condition in place, the ATS type checker is now able to successfully verify that the pre-condition for `read_data` is satisfied, namely that `i == p || i == t`. And thanks to the error

raised by the ATS type-checker, I was able dodge a subtle concurrency bug before I even wrote a single line of running code.

5.2 Case study: Model-checking a complex entry handler

In Chapter 2.4.1 I described the design principles behind the entry handler, as well as a simple entry handler. For more elaborate entry handlers, I have chosen to gain some assurance of their correct operation by using model-checking techniques. It is the case that model-checking for ATS programs (Ren, 2014) exists, but it is still a work-in-progress that may be suitable in the future. For the time being, I relied upon hand-coded assembly code and a partial encoding of semantics into the model-checking program called PAT: Process Analysis Toolkit (Sun et al., 2009). PAT supports a language called CSP# (Communicating Sequential Programs-Sharp) for modeling.

The more elaborate entry handler under review will be the one used by the `ehci` module. The goal of this entry handler is to manage two separate contexts: one for the "normal" thread of execution within the module, and a temporary context that is established for the purpose of handling IRQs. This entry handler can also be viewed as a specific implementation of a more general idea: managing multiple, preemptible threads using code written at the application level.

The transcription of the assembly code into the PAT model is shown below. The operation of the code is broken down into processes. A key mechanism in the model is the use of the internal choice operator `<>` to simulate the effect of interrupts: as we proceed from process to process, an internal choice is always offered back to `Start`. This simulates the possibility of having arbitrary restarts of the entry handler – at any moment. To help understand the meaning of the register assignments, please consult Table 5.1. A graphical diagram of the entry handler control flow is found in Figure 5.2, and a sample sequence of steps is shown in Figure 5.3 on page 143.

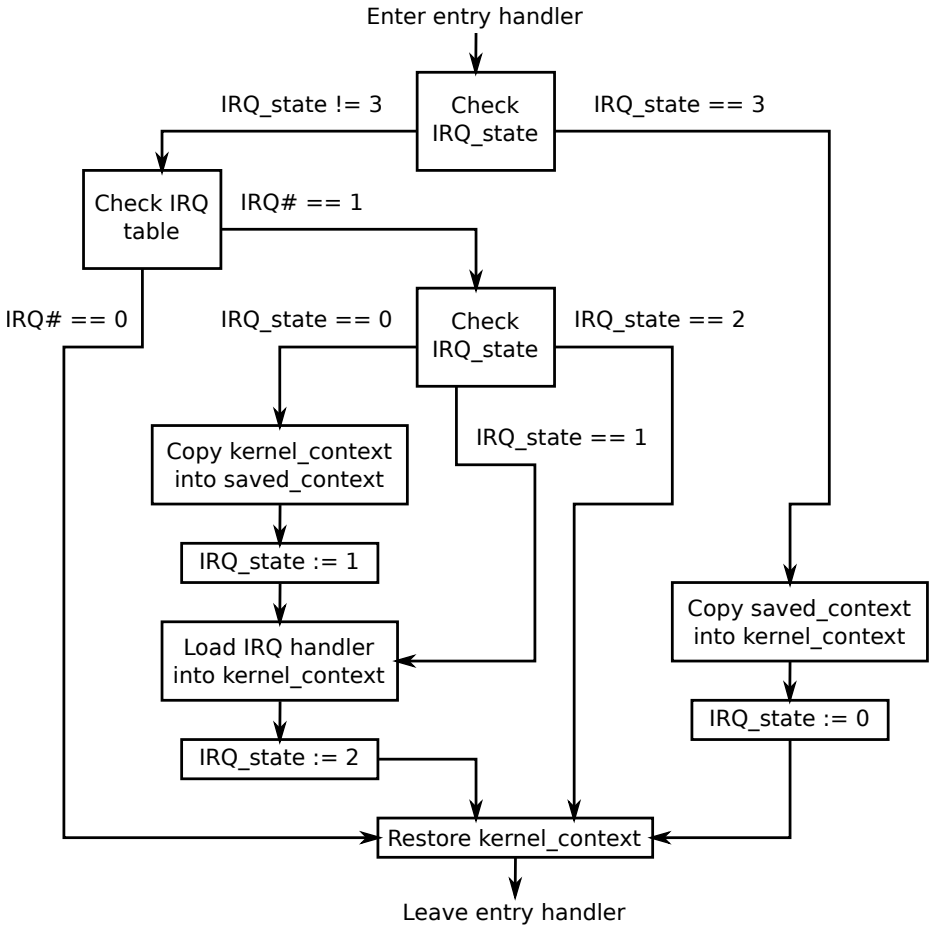

Figure 5.2: Entry handler control flow diagram

The first process, `InitEntry`, models the possibility of entering the program for the first time, when `r13` will be zero. Otherwise, the simulation continues under the presumption that `r13` will not be zero.

There are two possibilities for `InitEntry`, determined by internal choice. In the case of `firstTime`, we set up a bunch of model variables to match the conditions of entering the program for the first time. In the case of `normal`, the condition is simply

`r11`	Pointer to the IRQ state variable
`r12`	Pointer to the IRQ status table
`r13`	Pointer to the kernel-provided process context

Table 5.1: Meanings ascribed to registers in the `ehci` entry handler

	`InitEntry =`
`_start:`	`(firstTime {`
`CMP r13, #0`	`r13 = 0; ctxt = KernelCtxt;`
`LDREQ sp, =_stack`	`IRQstate = CSwitchStage; CSstep = 4;`
`LDREQ pc, =main`	`} -> Finish) <>`
	`normal { ctxt = KernelCtxt; } -> IRQ`

Table 5.2: The `InitEntry` stage

`LDR r11, =irq_state`	`Start = regularStart { r13 = r13address; } -> Start1;`
`LDR r1, [r11]`	`Start1 = if (IRQstate == RestoreStage) {`
`CMP r1, #3`	`RestoreSavedContext1 <> Start`
`BEQ 3f`	`} else { Start2 <> Start };`

Table 5.3: Check to see if we are returning from IRQ-handling mode

that the kernel has provided us a pointer to a valid context.

Then, the following process shown in Listing 5.1 simulates whether an IRQ is raised or not, or whether the IRQ handler has indicated that it has completed processing and would like to return.

Now, the real start of the interesting part of the entry handler is in Table 5.3, presuming that `r13` is equal to some non-zero address, and the first step is to test the IRQ state machine to see if the IRQ handler is ready to return. If so, we branch to `RestoreSaved` stage that is shown in Table 5.8.

Moving on, recall that the entry handler protocol defines `r12` as containing a valid pointer to an IRQ status table. Therefore we can check the status of IRQ 77 (in this case, for the `ehci` module) quickly, and branch appropriately. The `irq` variable models the result of querying the interrupt status table, as shown in Table 5.4.

Finally, in Table 5.5, we complete the initial case breakdown, picking up wherever the IRQ state said that we left off previously: either saving the "normal" context into

```
IRQ = (irqOff { irq = 0; } -> Start) <>
      (irqOn { irq = 1; } -> Start) <>
      (irqFinished { irq = 0; IRQstate = RestoreStage; } -> Start);
```

Listing 5.1: Simulating IRQs

```
LDRB r0, [r12, #77]  | Start2 = if (irq == 0) {
CMP r0, #1           |   ContextSwitch1 <> Start
BNE 2f               | } else { Start3 <> Start };
```

Table 5.4: Check the IRQ status table

```
                     | Start3 = if (IRQstate == SaveStage) {
                     |   SaveContext1 <> Start
                     | } else {
CMP r1, #1           |   Start4 <> Start
BEQ 1f               | };
BGT 2f               | Start4 = if (IRQstate == LoadIRQStage) {
                     |   LoadIRQContext1 <> Start
                     | } else {
                     |   ContextSwitch1 <> Start
                     | };
```

Table 5.5: Switch between the `Save`, `LoadIRQ`, or `ContextSwitch` stages

a designated space, loading the IRQ handling context, or completing the process of switching into it.

By this time you may have noticed that each step in the code is broken down as a "process" in the CSP# encoding. Those processes are then linked together, but at each step is inserted the possibility of an arbitrary branch back to the `Start`. That pattern continues for the `Save` stage, in Table 5.6.

Pairs of `LDM` and `STM` instructions are used to quickly copy memory from one address to another. The specific semantics of these instructions are only partially modeled. The `r13` variable tracks whether the original value is properly restored by the end. And the `SCstep` variable will be used to prove that the entire block executes atomically with respect to the overall entry handler. Finally, the IRQ state is advanced by storing the value of `1`, corresponding to the `LoadIRQ` stage, into the state variable addressed by `r11`.

The `LoadIRQ` stage in Table 5.7 creates a new IRQ-handling context by taking whatever existing context there is, clearing out the saved processor status register, and arranging the saved stack and program counter to point at the IRQ-handling

```
LDR r1, =saved_context
LDMIA r13!, {r2-r10}
STMIA r1!, {r2-r10}
LDMIA r13!, {r2-r10}
STMIA r1!, {r2-r10}
SUB r13, r13, #0x48

MOV r0, #1
STR r0, [r11]
```

```
SaveContext1 = sc1 { SCstep = 1; }
             -> (SaveContext2 <> Start);
SaveContext2 = sc2 { r13 = r13 + 36; SCstep = 2; }
             -> (SaveContext3 <> Start);
SaveContext3 = sc3 { SCstep = 3; }
             -> (SaveContext4 <> Start);
SaveContext4 = sc4 { r13 = r13 + 36; SCstep = 4; }
             -> (SaveContext5 <> Start);
SaveContext5 = sc5 { SCstep = 5; }
             -> (SaveContext6 <> Start);
SaveContext6 = sc6 { r13 = r13 - 72; SCstep = 6; }
             -> (SaveContext7 <> Start);
SaveContext7 = scDone { IRQstate = LoadIRQStage; }
             -> (LoadIRQContext1 <> Start);
```

Table 5.6: The `Save` stage

```
1:
MOV r0, #0
STR r0, [r13]
LDR r0, =ehci_irq_handler
STR r0, [r13, #0x44]
LDR r0, =_irq_stack
STR r0, [r13, #0x3C]

MOV r0, #2
STR r0, [r11]

B 2f
```

```
LoadIRQContext1 = lic1 { LICstep = 1; }
                -> (LoadIRQContext2 <> Start);
LoadIRQContext2 = lic2 { LICstep = 2; }
                -> (LoadIRQContext3 <> Start);
LoadIRQContext3 = lic3 { LICstep = 3; }
                -> (LoadIRQContext4 <> Start);
LoadIRQContext4 = lic4 { LICstep = 4; }
                -> (LoadIRQContext5 <> Start);
LoadIRQContext5 = lic5 { LICstep = 5; }
                -> (LoadIRQContext6 <> Start);
LoadIRQContext6 = lic6 { LICstep = 6; }
                -> (LoadIRQContext7 <> Start);
LoadIRQContext7 = licDone {
                  ctxt = IRQCtxt;
                  IRQstate = CSwitchStage;
                } -> (ContextSwitch1 <> Start);
```

Table 5.7: The `LoadIRQ` stage

stack and procedure. It then sets the IRQ state variable to 2, corresponding to the `CSwitchStage`, and branches to the context-switching stage.

The `RestoreSaved` stage in Table 5.8 occurs on the other end of the IRQ-handling procedure, when it is ready to return back to normal operation. Then, the previously-saved context needs to be copied back into the regular context area so that the context-switching mechanism can load it normally. Again, `LDM` and `STM` are used to quickly copy a 72-byte segment of memory, via spare registers that are used as scratch space. At the end, we assume the IRQ status was cleared by the handler, so the IRQ

```
3:
LDR r1, =saved_context
LDMIA r1!, {r2-r10}
STMIA r13!, {r2-r10}
LDMIA r1!, {r2-r10}
STMIA r13!, {r2-r10}
SUB r13, r13, #0x48

MOV r0, #0
STR r0, [r11]
```

```
RestoreSavedContext1 = rsc1 { RSCstep = 1; }
                     -> (RestoreSavedContext2 <> Start);
RestoreSavedContext2 = rsc2 {
                       r13 = r13 + 36; RSCstep = 2;
                     } -> (RestoreSavedContext3 <> Start);
RestoreSavedContext3 = rsc3 { RSCstep = 3; }
                     -> (RestoreSavedContext4 <> Start);
RestoreSavedContext4 = rsc4 {
                       r13 = r13 + 36; RSCstep = 4;
                     } -> (RestoreSavedContext5 <> Start);
RestoreSavedContext5 = rsc5 { RSCstep = 5; }
                     -> (RestoreSavedContext6 <> Start);
RestoreSavedContext6 = rsc6 {
                       r13 = r13 - 72; RSCstep = 6;
                     } -> (RestoreSavedContext7 <> Start);
RestoreSavedContext7 = rscDone {
                       irq = 0;
                       ctxt = SavedCtxt;
                       IRQstate = CSwitchStage;
                     } -> (ContextSwitch1 <> Start);
```

Table 5.8: The `RestoreSaved` stage

```
2:
LDR r0, [r13]
MSR cpsr_sf, r0
ADD r13, r13, #8
LDMIA r13, {r0-r15}
/* control flow ends */
```

```
ContextSwitch1 = cs1 { CSstep = 1; }
               -> (ContextSwitch2 <> Start);
ContextSwitch2 = cs2 { CSstep = 2; }
               -> (ContextSwitch3 <> Start);
ContextSwitch3 = cs3 { CSstep = 3; }
               -> (ContextSwitch4 <> Start);
ContextSwitch4 = cs4 { CSstep = 4; }
               -> Finish;

Finish = f {finish = 1;} -> Skip;
```

Table 5.9: The `ContextSwitch` stage

state returns back to 0, indicating that the normal context is back in place and things may proceed regularly. Then, assuming no interruption, we fall through to the context-switching stage.

Finally, the `ContextSwitch` stage, as shown in Table 5.9, is used to actually load and branch to the context that is stored in the memory space pointed to by `r13`. That is accomplished by loading the saved program status register from the first slot, and then using the `LDM` instruction to simultaneously load all 16 registers from the saved context. The ARM architecture defines any load to the program counter (`r15`) as an

```
#define cond finish == 1;
#define goal (r13 == 0 || r13 == r13address) &&
            (CSstep == 4) &&
            (!(LICstep == 0 && RSCstep == 0 && SCstep == 0) ||
              ctxt == KernelCtxt) &&
            (LICstep == 0 || (LICstep == 6 && ctxt == IRQCtxt)) &&
            (RSCstep == 0 || (RSCstep == 6 && ctxt == SavedCtxt)) &&
            (SCstep == 0 || (SCstep == 6 && ctxt == IRQCtxt)) &&
            (IRQstate == CSwitchStage || irq == 0);

#assert InitEntry |= [] (cond -> goal);
```

Listing 5.2: The property to be checked

immediate branching instruction, therefore, this instruction will load all registers and restore program flow in one swoop, as soon as it proceeds.

The model simulates this `LDM` instruction by excluding the internal choice from the final step. Upon executing the `LDM` the program counter is no longer within the entry handler, and therefore the assumptions about restartability no longer apply. The entry handler has effectively finished its job. Any interruptions that occur to the program outside of the entry handler will trigger the context-saving mechanism of the kernel, overwriting the kernel context space as normally indicated. The model contains an additional variable named `finish` that is used for the final assertion.

The final assertion, in Listing 5.2, is a Linear Temporal Logic (Pnueli, 1977) statement that `InitEntry` entails that it is always the case (`[]`) that the condition implies (`->`) the goal. The condition is simply that the Finish is reached. The goal is a complicated conjunction that can be broken down into several pieces:

- Either `r13` will be zero, or `r13` will be equal to the address (arbitrary pointer value) that it had upon entry.
- All four steps in the `ContextSwitch` stage will be executed.
- If none of the steps from the `Save`, `LoadIRQ`, or `RestoreSaved` stages are executed, then the final context must have come from the kernel.

- The `LoadIRQ` stage behaves atomically, and if it is executed then the final context must be the IRQ context.
- The `RestoreSaved` stage behaves atomically, and if it is executed then the final context must be the Saved context.
- The `Save` stage behaves atomically, and if it is executed then the final context must also be the IRQ context.
- Either the final IRQ state is the `ContextSwitch` stage, or there was no IRQ raised.

I consider the successful validation of this property to be evidence that the modeled entry handler's control flow has the desired properties necessary to be safely interruptible and restartable.

5.3 Case study: Invasive changes to a device driver

While writing device drivers and similar low level systems code, I often found it necessary to rewrite large pieces of code in order to integrate new knowledge that I had obtained, or optimizations that I wished to express. With ATS, it was quite common to find that the code worked well, without additional testing, once I had managed to satisfy the type-checker and gotten the code to compiler. I will describe one example here: optimizing the packet transmit path of a μIP-based driver.

A quick explanation: μIP is extremely small and uses just a single global buffer named `uip_buf` to store both the incoming packet and the eventual outgoing packet. Therefore the ATS code that interfaces with μIP uses two functions to store and load `uip_buf`, as shown in Listing 5.3.

The viewtype `usb_mem_vt` describes a special piece of memory that satisfies the properties necessary to be used for memory-mapped I/O by the USB host controller device. It is indexed by both location and by size. Therefore, the `copy_*_uip_buf`

```
fun copy_into_uip_buf {n, m: nat | n <= m && n <= UIP_BUFSIZE} {l: agz} (
  ! usb_mem_vt (l, m), uint n
): void
fun copy_from_uip_buf {m: nat} {l: agz} (
  ! usb_mem_vt (l, m)
): [n: nat | n <= m && n <= UIP_BUFSIZE] uint n
```

Listing 5.3: The `copy_*_uip_buf` functions

```
fun do_one_send {i: nat} (usbd: ! usb_device_vt (i)): void
```

Listing 5.4: The older, unoptimized signature for `do_one_send`

functions are safe because they will not allow more than `UIP_BUFSIZE` to be copied to or from the `uip_buf`, and the device memory space must be big enough for the copy as well.

The μIP library expects the programmer to provide a `do_one_send` function that takes the contents of the `uip_buf` and transmits it over the physical layer, in this case, Ethernet. Since μIP uses a global variable to hold the data, the original version of the `do_one_send` function was specified simply as shown in Listing 5.4.

This interface meant that the function had to allocate a URB as well as a piece of device-accessible memory every single time that μIP wanted to send a packet, and then release both resources. While not the hardest tasks to accomplish, doing this was completely unnecessary. I had already written the μIP main loop to keep around and reuse a URB and a buffer for the receive path. A slight advantage could be gained by also keeping a persistant URB and buffer for reuse by the transmit path. In addition, with the old method, the `do_one_send` function had to wait until the packet was successfully processed by the hardware before it could release the resources and return. With the new method, there is no need for `do_one_send` to wait, because it does not need to release the resources. Instead, it can return control back to the μIP main loop while the hardware sends the packet, and by the time control returns to `do_one_send`, the packet will be long gone. Of course, that the resource is available

```
fun do_one_send {i, ntTDs: nat} {tactive: bool} {tl: agz} (
  txfer_v: ! usb_transfer_status i |
  usbd: ! usb_device_vt (i),
  turb: ! urb_vt (i, ntTDs, tactive) >> urb_vt (i, ntTDs', tactive'),
  tbuf: ! usb_mem_vt (tl, USBNET_BUFSIZE)
): #[ntTDs': nat] #[tactive': bool] void
```

Listing 5.5: The newer, optimized signature for `do_one_send`

```
implement do_one_send (txfer_v | usbd, turb, tbuf) = let
  val _ = urb_wait_if_active (txfer_v | turb)
  val _ = urb_transfer_completed (txfer_v | turb)
  val _ = urb_detach_and_free_ (turb)
  val txlen = copy_from_uip_buf (tbuf)
  val header = ((txlen lxor 0xFFFF) << 16) lor txlen
  val _ = usb_buf_set_uint_at (tbuf, 0, header)
  val (xfer_v | s) =
    urb_begin_bulk_write (turb, txlen + 8, tbuf)
  prval _ = (txfer_v := xfer_v)
in () end
```

Listing 5.6: Implementation of `do_one_send` for the ASIX driver

is still required to be verified, but it takes relatively little time to do so. The new signature for the `do_one_send` function is presented in Listing 5.5.

The new interface is significantly busier. Obviously, we need to pass along an URB (named `turb`) and a buffer (named `tbuf`). Since a transmission could have been initiated by an earlier call, we need to pass along the view representing the current transfer status, so that it can be retired safely. Also, you can see that `do_one_send` makes no assumptions about the status of the URB: whether it has a chain of TDs, or whether it is active. It simply takes it as it is, and returns an updated version with a new transfer established. The implementation has been considerably simplified too, as a result of not having to acquire and release resources. It is short enough to be shown in Listing 5.6.

First, the function checks the old transfer to see if it is still running, which it most likely is not. Then it flushes out the old transfer and frees up the TDs. The new data is copied into the `tbuf` and its length is placed into the variable `txlen`. The

ASIX hardware requires that the first word of the USB transfer be a command that carries the length of the following packet twice: in the most significant 16 bits, and also in the least significant 16 bits. That value is prepared in the `header` variable and then stored into the first word of `tbuf` (which has been conveniently reserved for this purpose). Now, all the function has to do is kick off the bulk transfer and return the new `usb_transfer_status` view in place of the old.

All that's just the start. Now the μIP main loop must be refactored to allocate a transmit URB and buffer, as well as thread those two resources through all the paths leading to `do_one_send`. In addition, all of the exit paths must be updated with code to release these two new resources in case of an error. But it's just a matter of letting the compiler find the problems. All of the call-sites of `do_one_send` need to be updated with new parameters. But then those parameters have to come from further up the chain. So I insert new parameters to the callers, and I add some more resource allocation to the initialization function. Now the type-checker complains that these resources are not being released in several places. For each one, I go and insert the necessary code.

Originally, in `do_one_send`, I had used the function named `usb_wait_while_active` instead of the `usb_wait_if_active` that is shown in Listing 5.6. But that caused the type-checker to print out an error: it could not prove that the number of TDs for `turb` was greater than zero. Since `usb_wait_while_active` assumes that the TD chain exists, the function type has a guard that requires the number of TDs to be provably positive. The function `usb_wait_while_active` would cause a data abort error if allowed to run on a URB without TDs. After some thought, I realized that, in fact, it is possible that this new URB reserved for the transmit path may not always have TDs attached. So it would not be correct to use `usb_wait_while_active` every time on `turb`. Thankfully, the type-checker prevented that crash bug before it ever happened. Then, I simply wrote a function named `usb_wait_if_active` that is safe to use no matter if there are no

TDs attached, and avoided the problem altogether. Altogether, the newly refactored code was up and running and responding to packets correctly the first time it was booted.

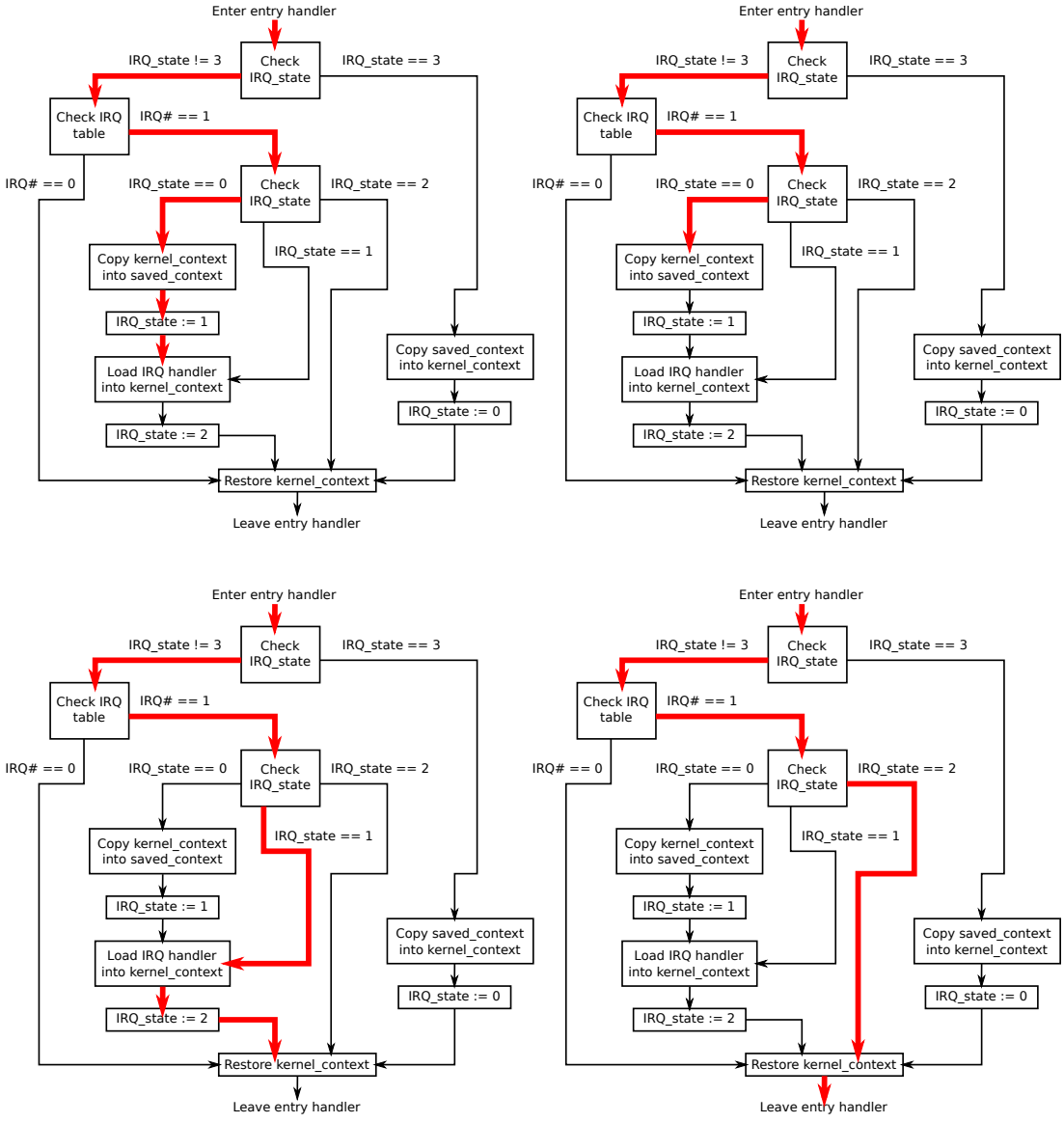

Figure 5.3: Entry handler control flow example, with interrupts forcing three restarts

Chapter 6

Conclusion

6.1 Critiques

There are some differences between functionality available to hardware and software that restrict the programming model the kernel can provide. For example, hardware can be wired to react to changes in individual bits. Terrier can use the delivery of software-invoked interrupts to achieve some of this functionality, but it can never approach the nanosecond level of response that hardware can exhibit.

The decision to use reentrant, restartable entry handlers instead of interrupt masking means that programs are susceptible being interrupted by events that are not directly relevant to their operation. This choice was made because in a real-time system, all interrupts do have some relevance to all programs, insofar as they consume time to handle, and therefore may influence the remaining program behavior

The choice of static priority, rate monotonic scheduling does lead to some loss of flexibility and, in some cases, capacity. Application choices of capacity and period must be evaluated on a case-by-case basis.

6.2 Future work

Terrier is not a completed system, and likely never will be. Such is the nature of operating system projects. There is always something more to be done. However, aside from additional hardware support, there are several avenues that might be explored given the opportunity:

- Expand the use of higher-order functions to help programmers write more elegant, easier code.

- Using higher-order functions, provide low-level support for an event-based programming paradigm such as functional reactive programming.
- Incorporate the use of session types for better reasoning about concurrency and communication.
- Design a method for explicit static reasoning about temporal constraints and use it for static verification of worst-case execution bounds.
- Investigate the use of other scheduling algorithms than basic rate monotonic scheduling.

References

E. Alkassar, M. A. Hillebrand, D. Leinenbach, N. W. Schirmer, and A. Starostin. The Verisoft Approach to Systems Verification. In *VSTTE '08: Proceedings of the 2nd international conference on Verified Software: Theories, Tools, Experiments*, pages 209–224, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-87872-8. doi: http://dx.doi.org/10.1007/978-3-540-87873-5_18.

J. Anderson, R. N. Watson, D. Chisnall, K. Gudka, I. Marinos, and B. Davis. TESLA: temporally enhanced system logic assertions. In *Proceedings of the Ninth European Conference on Computer Systems*, page 19. ACM, 2014.

ARM Architecture Reference Manual. ARM Limited, ARMv7-A and ARMv7-R edition, 2011.

M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal methods for Components and Objects*, pages 364–387. Springer, 2006.

S. Basumallick and K. Nilsen. Cache Issues in Real-Time Systems, 1994.

A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on operating systems principles*, pages 29–44. ACM, 2009.

B. N. Bershad, D. D. Redell, and J. R. Ellis. Fast Mutual Exclusion for Uniprocessors. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*, pages 223–233, 1992.

B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 267–284, 1995.

L. Cardelli et al. Modula-3 report (revised). Technical report, Digital Equipment Corp. (now HP Inc.), Nov 1989.

J. Chen and A. Burns. A fully asynchronous reader/writer mechanism for multiprocessor real-time systems. Technical report, University of York, 1997.

J. Chen and A. Burns. Asynchronous data sharing in multiprocessor real-time systems using process consensus. In *Proceedings of the 10th Euromicro Workshop on Real-Time Systems*, pages 2–9, 1998.

J. Chen and A. Burns. Loop-free asynchronous data sharing in multiprocessor real-time systems based on timing properties. In *Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications*, pages 236–246, 1999. doi: 10.1109/RTCSA.1999.811236.

A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. *ACM SIGPLAN Notices*, 46(6):234–245, 2011.

M. Danish and H. Xi. Using lightweight theorem proving in an asynchronous systems context. In J. M. Badger and K. Y. Rozier, editors, *NASA Formal Methods*, volume 8430 of *Lecture Notes in Computer Science*, pages 158–172. Springer International Publishing, 2014.

L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

D. Deville, C. Rippert, and G. Grimaud. Trusted Collaborative Real Time Scheduling in a Smart Card Exokernel. Rapport de recherche RR-5161, INRIA, 2004. URL `http://hal.inria.fr/inria-00077048`.

A. Dunkels. uIP: A tiny TCP/IP stack for embedded systems. `https://github.com/adamdunkels/uip`, 2013.

K. Elphinstone and G. Heiser. From L3 to seL4: What Have We Learnt in 20 Years of L4 Microkernels? In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 133–150, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8.

D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the fifteenth ACM symposium on operating systems principles*, pages 251–266, New York, NY, USA, 1995. ACM. ISBN 0-89791-715-4. doi: http://doi.acm.org/10.1145/224056.224076.

T. Hallgren, M. P. Jones, R. Leslie, and A. Tolmach. A principled approach to operating system construction in Haskell. *SIGPLAN Notices*, 40(9):116–128, 2005. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1090189.1086380.

C. Hawblitzel and E. Petrank. Automated verification of practical garbage collectors. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 441–453, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-379-2. doi: 10.1145/1480881.1480935. URL `http://doi.acm.org/10.1145/1480881.1480935`.

C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad Apps: End-to-end security via automated full-system verification. In

USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2014.

N. Henderson and S. Paynter. The formal classification and verification of Simpson's 4-slot asynchronous communication mechanism. In L.-H. Eriksson and P. A. Lindsay, editors, *FME 2002: Formal Methods - Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 350–369. Springer Berlin Heidelberg, 2002. ISBN 978-3-540-43928-8.

H. Herbelin. Coq proof assistant. `http://coq.inria.fr/`, 2009.

M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

M. Hohmuth and H. Tews. The VFiasco approach for a verified operating system. In *Proceedings of the 2nd ECOOP Workshop on Programming Languages and Operating Systems*, 2005.

G. C. Hunt and J. R. Larus. Singularity: Rethinking the Software Stack. In *ACM SIGOPS Operating System Review*, volume 41, pages 37–49. Association for Computing Machinery, Apr 2007.

E. H. Jensen, G. W. Hagensen, and J. M. Broughton. A new approach to exclusive data access in shared memory multiprocessors. Technical report, UCRL-97663, Lawrence Livermore National Laboratory, 1987. URL `https://e-reports-ext.llnl.gov/pdf/212157.pdf`.

T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A Safe Dialect of C. In *Proceedings of the USENIX Annual Technical Conference*, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association. ISBN 1-880446-00-6.

G. Klein, K. Elphinstone, G. Heiser, et al. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, USA, Oct 2009.

K. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, 2010.

Y. Li, R. West, and E. Missimer. A virtualized separation kernel for mixed criticality systems. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 201–212. ACM, 2014.

J. Liedtke. On micro-kernel construction. In *Proceedings of the fifteenth ACM symposium on operating systems principles*, pages 237–250, New York, NY, USA, 1995. ACM. ISBN 0-89791-715-4. doi: http://doi.acm.org/10.1145/224056.224075.

C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM*, 20(1):46–61, Jan. 1973. ISSN 0004-5411. doi: 10.1145/321738.321743. URL `http://doi.acm.org/10.1145/321738.321743`.

S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 103–116. ACM, 2007.

G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, 2005. doi: http://doi.acm.org/10.1145/1065887.1065892.

G. Parmer and R. West. Predictable Interrupt Management and Scheduling in the Composite Component-Based System. In *Real-Time Systems Symposium, 2008*, pages 232–243, 2008. doi: 10.1109/RTSS.2008.13.

L. C. Paulson. *Isabelle*, volume 828. Springer, 1994.

S. Peyton-Jones, S. Marlow, et al. The Glasgow Haskell Compiler. `http://www.haskell.org/ghc/`, 2014.

A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th annual symposium on foundations of Computer Science*, pages 46–57, Oct 1977. doi: 10.1109/SFCS.1977.32.

Z. Ren. Model Checking ATS. `http://ats-documentation.readthedocs.org/en/latest/conats/content.html`, 2014.

D. M. Ritchie and K. Thompson. The UNIX Time-sharing System. *Commun. ACM*, 17(7):365–375, July 1974. ISSN 0001-0782. doi: 10.1145/361011.361061. URL `http://doi.acm.org/10.1145/361011.361061`.

J. Rushby. Model checking Simpson's four-slot fully asynchronous communication mechanism. *Computer Science Laboratory–SRI International*, 2002.

J. M. Rushby. Design and verification of secure systems. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, SOSP '81, pages 12–21, New York, NY, USA, 1981. ACM. ISBN 0-89791-062-1. doi: 10.1145/800216.806586. URL `http://doi.acm.org/10.1145/800216.806586`.

L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, Sept. 1990. ISSN 0018-9340. doi: 10.1109/12.57058. URL `http://dx.doi.org/10.1109/12.57058`.

H. Simpson. Four-slot fully asynchronous communication mechanism. In *IEE Proceedings, Vol 137, Pt. E, No. 1*. IEE, Jan 1990.

H. Simpson. Correctness analysis for class of asynchronous communication mechanisms. *IEE Proceedings E (Computers and Digital Techniques)*, 139:35–49, 1992.

H. Simpson. Multireader and multiwriter asynchronous communication mechanisms. *IEE Proceedings - Computers and Digital Techniques*, 144:241–243(2), July 1997a. ISSN 1350-2387.

H. Simpson. Role model analysis of an asynchronous communication mechanism. In *Computers and Digital Techniques, IEE Proceedings*, volume 144, pages 232–240. IEE, 1997b.

H. Simpson. Protocols for process interaction. *IEE Proceedings on Computers and Digital Techniques*, 150(3):157–182, May 2003. ISSN 1350-2387. doi: 10.1049/ip-cdt:20030419.

S. Sridhar, J. S. Shapiro, and S. F. Smith. Sound and complete type inference for a systems programming language. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, pages 290–306, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-89329-5. doi: http://dx.doi.org/10.1007/978-3-540-89330-1_21.

U. Steinberg, J. Wolter, and H. Hartig. Fast component interaction for real-time systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 89–97, 2005. doi: 10.1109/ECRTS.2005.16.

J. Sun, Y. Liu, J. S. Dong, and J. Pang. Pat: Towards flexible verification under fairness. In *Proceedings of the 21th International Conference on Computer Aided Verification (CAV'09)*, volume 5643 of *Lecture Notes in Computer Science*, pages 709–714. Springer, 2009.

L. Torvalds et al. The Linux Kernel Foundation. `http://www.linuxfoundation.org/`, 2014.

R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Worst-case Execution-time Problem—Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.*, 7(3): 36:1–36:53, May 2008. ISSN 1539-9087. doi: 10.1145/1347375.1347389. URL `http://doi.acm.org/10.1145/1347375.1347389`.

H. Xi. Applied Type System (extended abstract). In *Post-workshop Proceedings of TYPES 2003*, pages 394–408, 2004.

J. Yang and C. Hawblitzel. Safe to the Last Instruction: Automated Verification of a Type-safe Operating System. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 99–110, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0019-3. doi: 10.1145/1806596.1806610. URL `http://doi.acm.org/10.1145/1806596.1806610`.

CURRICULUM VITAE

Matthew Danish
born 1982

Boston University – MCS 138 – Computer Science Department
111 Cummington Mall
Boston, MA 02215 USA

Phone: +1 617-870-4182
Email: md@bu.edu
URL: http://cs-people.bu.edu/md

Education

May 2004 B.S. in Logic and Computation, Carnegie-Mellon University.
May 2015 Ph.D. in Computer Science, Boston University.

Other positions

2004—2008 Research Programmer at Intelligent Coordination and Logistics Laboratory, Robotics Institute, Carnegie-Mellon University, Pittsburgh, PA, USA.
Summer 2011 Internship at VMWare, Palo Alto, CA, USA.

Publications and talks

Papers

2010 Matthew Danish and Hongwei Xi. Operating system development with ATS. In Proceedings of the 4th Workshop on Programming Languages meets Program Verification. Madrid, Spain.

2011 Matthew Danish, Ye Li and Richard West. Virtual-CPU scheduling in the Quest operating system. In Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium. Chicago, IL, USA.

2011 Ye Li, Matthew Danish and Richard West, Quest-V: A virtualized multikernel for high-confidence systems. Technical Report: arXiv:1112.5136, arXiv.org. Also BU Technical Report, 2011-029, Boston University.

2014 Matthew Danish, Hongwei Xi. Using lightweight theorem proving in an asynchronous systems context. In Proceedings of the Sixth NASA Formal Methods Symposium. Houston, TX, USA.

Talks and posters

2012 Matthew Danish, Hongwei Xi and Richard West. Applying language-based static verification in an ARM operating system. In Work-in-Progress Poster Session of the 33rd IEEE Real-Time Systems Symposium. San Juan, PR, USA.

2013 Matthew Danish. Applying language-based static verification in an ARM operating system. Talk presented at the High Confidence Software And Systems Conference. Annapolis, MD, USA.

2013 Matthew Danish. Functional Pearl: Four slot asynchronous communication mechanism. Presented at the ACM SIGPLAN Workshop on Dependently-Typed Programming, informal work-in-progress talk session. Boston, MA, USA.

Teaching

Fall 2009 Assistant for CS131 (Combinatorial Structures)
Spr 2010 Assistant for CS131 (Combinatorial Structures)
Fall 2012 Assistant for CS108 (Application Programming)
Fall 2014 Assistant for CS330 (Algorithms)
Spr 2015 Assistant for CS131 (Combinatorial Structures)

Software

2009—2011 Collaborated on Quest-V: Virtualized separation kernel for mixed criticality systems.

2011—2012 Puppy: A barebones framework for operating system development on ARM.

2012—2015 Terrier: A real-time, embedded operating system for ARM using types for safety.