Toward Automatic Operating System Ports via Code Generation and Synthesis

Dissertation Advisors:
Margo I. Seltzer
Stephen Chong

Author:
David Andrew Holland

**Toward Automatic Operating System Ports
via Code Generation and Synthesis**

Abstract

Porting operating systems is expensive. Recent developments in formal specifications of machine architectures and in program synthesis have made it possible to consider generating code for operating system ports rather than needing to write it by hand. Several challenges arise, the foremost being the current scaling limits of program synthesis for assembly language.

In this dissertation I take a system-level approach to generating ports of a small operating system. I present Aquarium, an ecosystem of languages and tools I designed to surround an assembly language synthesis engine. Chief among these are Grayling, a special-purpose compiler for context switch code, and Grouper, a tool implementing an approach I call *compositional code generation*, generating code in units of small synthesizable blocks. I have deployed these tools in the context of a simple though realistic operating system called Goldfish, and successfully generated working code for a number of machine-dependent operating system components.

Contents

List of Figures

Acknowledgments

Thanks to my advisors and committee: Margo Seltzer, Stephen Chong, and James Mickens.

Also thanks to my colleagues on the PRINCESS project: Jingmei Hu, Eric Lu, Abby Lyons, and Ming Kawaguchi. James Bornholt helped us with many synthesis and solver issues.

Michal Kolesár provided this LaTeX template and Yihe Huang collaborated on TeX and dissertation-formatting issues.

Sam Fishman wrote the first ARM port of OS/161 that in some ways enabled the entire project.

Margo Seltzer arranged funding for years to support the development and ongoing maintenance of OS/161, without which this project and several others would not have been possible.

Thanks also to the many colleagues who have helped teach CS161 and teach with OS/161 over the years; these include, but are not by any means limited to, Sasha Fedorova, Ada Lim, Anne Solmssen, Daniel Margo, Jay Moorthi, Alex Patel, and Geoffrey Werner-Challen, not to mention the faculty who have taught here and elsewhere with OS/161.

Colleagues and friends in NetBSD have often helped out with advice and tidbits about real problems with porting to real hardware, as well as other technical issues, and also listened while I vented; Taylor Campbell, Alistair Crooks, Matthew Green, Jared McNeill, Maya Rashish, Mindaugas Rasiukevicius, Joerg Sonnenberger, S.P. Zeidler, and Christos Zoulas come particularly to mind but there are many others.

The programming languages group put up with my systems arrogance when I first ventured across the divide and eventually took me in when my advisor left for another school. Norman Ramsey first taught me about type systems; Greg Morrisett first taught me formal programming language semantics; Steve Chong taught me about program analysis. Gregory Malecha in particular put up with my silly questions for many years. Other neighbors (and friends) in the department, far too many to name, have been helpful and supportive over the years in myriad ways.

Thanks to friends and family, who let me complain to them at length and then send me food and books – Ada, David, Eric, Norm, Steve, Susan, Uri and many others.

Many years ago Laird Popkin and Kee Hinckley ran bulletin boards that played a critical role in setting high-school me on a better path than becoming a tech bro. Kathy Lampert first tried to coach me at public speaking, a thoroughly unrewarding but absolutely vital task.

A special final thanks to all my students over the years (in CS161 and otherwise), who I may not name; you have kept me honest and forced me to refine my understanding.

For my late father, Jerome T. Holland, who walked this road alone.

Chapter 1

Introduction

Porting operating systems is expensive. Bringing up an existing, portable operating system on a new hardware platform requires a lot of code; when the new hardware platform involves a new processor architecture this becomes even more code. For example, in NetBSD 9.0 there are approximately 39000 lines of code in 259 files specific to 64-bit ARM (“aarch64”), excluding compiler and toolchain, and not counting comments and blank lines. Writing this code not only takes time and money; it consumes the attention of experienced developers who might otherwise be free to pursue other goals, such as security or scalability.

Automatically generating this code is an attractive proposition. Several trends have combined recently to make this also a plausible proposition. First, advances in formal methods and verification have made it plausible to write down and work with formal specifications of real machine architectures [2], and CPU vendors are increasingly involved in producing and maintaining these specifications. Second, advances in program synthesis [4, 5, 9, 11, 22, 30, 31] have made it possible to generate small code chunks purely from specifications in reasonable amounts of time. Meanwhile, after a long period of almost monoculture, portability is becoming important again [12].

Machine-dependent operating system code has particular properties that make it both an attractive target and a difficult target for such a venture. Operating systems are structured to be portable; the code needed for a new port consists of relatively small pieces, many of them mutually disjoint, and all with relatively clear interfaces. Those interfaces will, in fact, have machine-independent specifications: the points at which machine-dependent code meets machine-independent code are necessarily uniform, or close to it, across machine architectures. On the other hand, many of these pieces must be written in assembler (because they do things that cannot be expressed at higher levels), many are delicate and difficult or impossible to debug if incorrect, and while they may each be small relative to the total size, or the size of a whole operating system, they may still be large relative to the scaling limits of program synthesis. Meanwhile, synthesizing assembly language turns out to itself be inherently difficult [15].

This dissertation concentrates on deploying code synthesis in an operating system and producing important parts of ports to multiple architectures. It examines the hypothesis that progress in program synthesis, and in particular assembly language synthesis, makes it possible to automatically generate the machine independent parts of an operating system. It is part of a larger project, whose goal was itself a subset of the complete problem of generating operating system ports. The larger project's target is what one might call CPU control code: code that manipulates CPU state and implements actions that must be done differently on different architectures. This excludes a number of other things equally necessary for complete ports. Most notably, this excludes the compiler, which is a large project in its own right [8, 23]. It also excludes device drivers, which are a different kind of code and a different problem. (A good way to think about the difference: in CPU control code, one generally knows what to do, because that is the specification, and the goal is to figure out how; in device drivers, generally the goal is to figure out what to do, and then how to do it is relatively straightforward and can even be filled in by hand afterward [27].) We also excluded system board issues such as interrupt routing and DMA configuration, less for any particular technical reason than because collaborators were already working on this [1, 17]. Note that a substantial amount of code remains after these exclusions, including all the machine-dependent parts of the virtual memory system.

There have been a number of challenges. First and foremost, synthesis has scaling limits. Our project's synthesis engine can generate assembly blocks of up to four or sometimes five instructions, and with additional deduction rules up to twelve for certain well-structured specifications [16]. Unfortunately, while this is enough for some machine-dependent operating system elements, others are larger. For example, the MIPS general exception handler in NetBSD 9 is about 100 instructions.

Second, one must develop specifications for the full range of machine-dependent elements in a full operating system port. Even excluding the parts outside the project scope, this has generated a number of further challenges that needed to be met. For example, I needed to develop a model for the RAM cache that is complete enough to allow the synthesis engine to issue cache control instructions but simple enough to avoid swamping it with irrelevant architectural complexity.

Third, these specifications must be machine-independent; they may not, for example, mention specific control registers. Instead they must be written in terms of abstract machine state, and some mechanism is required to instantiate these abstractions for each machine architecture. This is reasonable enough, though not precisely trivial, for machine-dependent code called from machine-independent code, which does things the machine-independent code has some model for. It becomes more problematic for code invoked directly by the hardware, which may need to reason about concepts that are not just machine-dependent but machine-*specific*. This arises both in the kernel's startup code (the code that runs before the kernel reaches `main`) and also in some exception handlers.

Fourth, one must interact with the operating system's machine-independent code base. The project's synthesis engine produces assembly code; it does not produce any of the other things needed in an operating system port, such as header files with declarations required by the machine-independent code, or data structures to put in those header files, to say nothing of makefiles, kernel configuration files, linker scripts, or other miscellania.

Finally, one must collect up the independent and sometimes disparate solutions to these assorted problems and forge them into a system where all the components interoperate as needed.

For this dissertation I have taken a system-level approach to these problems. Taking our project's synthesis engine as a building block, I have designed and developed a collection of other tools and techniques that, acting in concert, allow me to address a significant fraction of the challenges.

1.1 Summary of Approaches

In the previous section I identified five families of challenges involved in generating operating system ports. This section briefly summarizes the approaches I have taken to each.

1.1.1 Scaling

Scaling limits of our synthesis engine preclude generating whole assembly language procedures at once. I have chosen to embrace this limit (rather than struggle against it) and make the basic unit of code generation sub-procedure units (we call them "blocks"). This requires tools and infrastructure for composing these blocks into full procedures, and for handling specifications so as to make the composition reasonably safe. I have developed a technique I call "compositional code generation" for this purpose and built a tool called Grouper (Chapter 7) to implement it.

It also becomes reasonable, in fact desirable, to support multiple methods of block generation. Synthesis is flexible, but not always performant; compilation is relatively performant, but not always flexible enough for low-level code. Applying both techniques together permits synergies. Furthermore, the small scale and potentially broad interfaces of single blocks permit the development of special-purpose block compilers.

I have identified one common target for such a tool (the bulk register saves done by operating system code to save and restore machine context, which are both potentially large and hard to synthesize) and developed an implementation called Grayling (Chapter 6).

I identified two other related common targets (blocks for making function calls, and the prologue and epilogue blocks of procedures) that are difficult to specify in longhand (because calling conventions can be complicated), and committed a hack to extract this code from a C compiler without needing direct knowledge of the details. This is called Nautilus and is described briefly in Section 5.6 of Chapter 5.

I did *not* write a block generator that compiles from a higher-level language designed to interact with the other tools; this would be a useful piece of future work.

Blocks can also be written by hand and presented as canned chunks.

1.1.2 Specifications

My approach to writing workable specifications for complex things (such as the cache, as already noted) is to take advantage of a powerful shortcut: the specification does not need to describe everything that happens. It only needs to cause the synthesis instruction to pick the right instruction with the right operands. The cache model is described (as an example of a general technique) in Section 8.3.12.

There are also cases, such as memory barriers, where I have concluded that the cost of writing specifications and providing suitably expressive specification logics is not worthwhile, at least in the near term. Thus, implementations of these elements are to be provided as canned code blocks by the person preparing the port. (This is discussed in Chapter 4.)

To help cope with the sheer number of machine-dependent elements and specifications, as well as minimize the problems caused by the OS code base, I have used OS/161 [14] as my test platform. OS/161 is a teaching OS; it is small compared to production systems but also realistic. It also has had a lot of work put into making its internal abstractions clean, and while it natively only supports MIPS on its own emulator, System/161, it has been successfully ported to other platforms. (Read more in Chapter 8.)

1.1.3 Machine Independence

I have two approaches for writing machine-independent specifications: *refinement* and *predicate abstraction*. “Refinement” means dividing the problem into multiple possible approaches (forming some sort of covering set, rather than one per machine); for each port one specifies which approach to take and possibly values for approach-specific parameters. This is, for example, the usual way of thinking about endianness. “Predicate abstraction” means writing the problem specification in terms of fully abstracted elements (constants, machine state, predicates on machine state, etc.); for each port one specifies concrete substitutions for each such element. For example, interrupt control operations can be specified in terms of an abstract `interrupts_on` predicate, which can then be concretized to refer to whatever control register bits or internal processor state controls this on any given machine.

For machine-*specific* code, if any magic bullet exists I have not found it. The best way forward seems to be to write machine-dependent specifications for the machine-specific code. I have left this for future work; my tools and environment were set up optimistically and do not support machine-dependent specification material.

1.1.4 Generating Other Material

To help cope with the need to generate data structures, Grayling can generate C definitions for the context structures it operates on. So far this has proven sufficient to deal with all such needs; an additional more general tool capable of synthesizing data structure definitions was planned, but it has not been needed and thus has not been written.

I also wrote a simple templating tool called Hermitcrab (discussed briefly in Chapter 5) that interoperates with all the other tools and can insert definitions or generated code into arbitrary output files. This allows producing necessary components such as header files and makefile fragments.

1.1.5 System Design

To facilitate interoperation all my tools use a common interchange language called Tuna, so named because it originated as the vehicle for delivering canned code chunks.

I decided that for visibility and debuggability reasons that all tools had to run "offline" – one runs them, they read their inputs, they create their outputs. Even where it might be more natural, tools do not directly interact or execute one another. This has been moderately painful, but also invaluable for debugging and for forcing clarity upon the indirect interactions that do happen.

1.2 Contributions

Consequently, the contributions of this dissertation are:

- The design of a language and tool ecosystem, Aquarium, to facilitate the practical use of code synthesis. (Chapter 5).
- A special-purpose compiler, Grayling, for context switch code, which has traditionally needed to be written in assembler. (Chapter 6).
- A code generation technique (compositional code generation) based on composition of small heterogeneous blocks, and a tool, Grouper. (Chapter 7).
- A deployment of program synthesis in a realistic operating system setting. (Chapter 8.)

and on a lesser scale,

- A set of lightweight models for various low-level phenomena at the interaction of the machine and the OS, most notably the RAM cache, useful for synthesis and likely also verification.
- Formal specifications for many of the machine-dependent portions of the OS.
- An implementation.

Note that the generator for blocks related to calling conventions (Nautilus, described briefly in Section 5.6) is *not* a contribution.

1.3 Thesis Statement

Despite the scaling limits of program synthesis and assembly language synthesis in particular, given suitable tooling, synthesis is a viable technique for generating real code that does useful things, in particular significant portions of the machine dependent code that operating systems rest on.

1.4 Terminology and Notation

In this dissertation I have endeavored to use standard terminology and notation, but that is not always entirely possible and for some things clarification may be helpful. Several items of OS terminology (including "context switch") are clarified in Chapter 2. However, I would like to first define what I mean by *machine*.

A machine or machine architecture is an abstract specification for a CPU or for a family of related CPUs. It defines a set of instructions, and also registers, control registers, operating modes, and functional models for hardware-level abstractions (caches, memory, memory management, etc.) I have chosen to avoid the largely equivalent term "instruction set architecture" because it is not always understood to include all these elements. Most machine specifications have evolved over time, with multiple revisions and variants appearing.

Code (or specifications, or designs, or whatever else) is *machine-dependent* if it refers to or relies on properties of a machine that are particular to that machine, and *machine-independent* if it does not. In all cases, however, this is relative to some (usually implicit) concept of what properties are universal among reasonable or interesting machines. For example, there are many common machines where registers are 32 bits wide and many where registers are 64 bits wide. Code that assumes one or the other width is therefore machine-dependent. However, code that assumes the width of registers is a power of two, or that memory is subdivided into 8-bit bytes, is in most contexts considered adequately machine-independent even though historically there have been many 36-bit word-addressed machines. Few operating systems today have machine-independent code that could be expected to work on a PDP-10 even if it fit into memory.

This dissertation distinguishes *machine-specific* from *machine-dependent*. Code is machine-dependent if code to do the same thing would be different on a different machine; it is machine-specific if code to do the same thing would be nonsensical or impossible on another machine. That is, machine-dependent code relies on properties or abstractions that vary from machine to machine but exist more or less universally. Machine-specific code relies on properties or abstractions that do not even exist on other machines. SPARC register windows are a classic example of a machine-specific phenomenon, and the kernel code needed to support them is entirely machine-specific. The distinction between “machine-dependent” and “machine-specific” has proven useful, but may itself be specific to the dissertation.

Another piece of terminology describes OS developers: I use *porter* for the person creating a new OS port and *OS architect* for the person who designed and implemented the machine-independent parts of the OS. (These are not necessarily exactly one person, of course; they may even be the same person. But they are different roles that it is helpful to distinguish.)

1.5 Document Organization

The remainder of this dissertation is structured as follows: Chapters 2 and 3 provide background material about operating systems and about program synthesis and verification, respectively. Chapter 4 analyzes the problem in more detail. Chapter 5 describes the ecosystem of tools and languages I and my colleagues on the project have developed. Chapters 6 and 7 cover Grayling, the context switch compiler, and Grouper, the block composition tool. Chapter 8 discusses the deployment in an operating system called Goldfish. Finally, Chapter 9 discusses some broader points and concludes.

Chapter 2

Operating Systems Background

This chapter provides quick background on, and clarification of terminology for, a few operating systems concepts for the benefit of readers from other communities. It goes into more detail for RAM caches, as this material is typically not covered in sufficient depth in textbooks or classes and good summaries are difficult to find. The final section (Section 2.5) describes, for context, the operating systems mentioned in this dissertation.

2.1 Terminology

2.1.1 CPUs vs. Cores

Today a microprocessor may have many cores, and often each core may have more than one "thread" that can execute independently. (These are not to be confused with software-level threads). Meanwhile, a system may have one or more "sockets" to plug microprocessor chips into. This can seem complicated, but for most software purposes, the hierarchy of sockets and cores and hardware threads is irrelevant. A CPU ("central processing unit"), from the operating system's point of view, is a piece of hardware that executes a single stream of instructions. Each hardware-level context (whether a core or one of several threads in a core) normally has its own register file, control registers, MMU (memory management unit), and is, to software running on it, a complete implementation of its architecture specification. Therefore one can, and I do, refer to all hardware execution contexts as "CPUs". A system has some number of them in total, usually not fixed at kernel compile time. Typically one is distinguished as the "boot" or "bootup" CPU that handles system and OS initialization and then starts the others. In a few cases, they may not all have the same capabilities. (They may not all even be the same machine architecture.) But even then, for operational purporses they all execute instruction streams and are effectively identical.

2.1.2 Mainboards and SoCs

A SoC ("system on a chip") is both a processor (with some number of CPUs, per the previous section) and what one might call a "mainboard" or "system board", containing additional hardware for communication and storage (and other purposes) and one or more system buses to connect that hardware to the processor. Today many computer systems are SoCs. To software there is no material difference between a SoC and a microprocessor on a traditional system board with discrete components and perhaps expansion slots. In this dissertation I refer to all of these as "mainboards".

The important difference is that while a CPU implements a machine architecture specification, a mainboard *contains* CPUs (sometimes heterogeneous models or even heterogeneous architectures) and many other devices. A complete port of an OS for a particular mainboard design involves not just porting to the machine architecture (or architectures) of the CPUs on it, but also writing drivers for the various devices, and often also board-specific code for finding the devices (or the controllers for the buses the devices are connected to), locating main memory, and so forth. As discussed in Chapter 1, both device drivers and this sort of mainboard-specific code are beyond the scope of both this dissertation and its surrounding project.

A *platform*, at least in this dissertation, refers to the combination of a CPU and a mainboard.

2.1.3 Context Switching

A *context switch* is when a CPU stops executing one instruction sequence and starts executing another, where the jump in control flow is in some way more substantive than an ordinary branch or call operation. There is no good fixed definition of the term and experts sometimes disagree about which operations "count" as context switches. This dissertation does not attempt to resolve the issue or provide a definition. Rather, I observe that most of the things usually or sometimes described as context switches involve saving and restoring CPU state. For example, switching from one thread of control to another, while preserving the illusion that each thread has exclusive access to the CPU, requires saving the values the first thread had in machine registers and restoring the values belonging to the second thread, saved at some previous point. Otherwise, the illusion is lost and incorrect executions occur. Similarly, handling an interrupt requires saving the interrupted state so that it can be restored after processing the interrupt. Different sets of registers must be saved depending on the operation. For the purposes of this dissertation, a *context operation* is either saving or restoring the values of some set of machine registers, a *context block* is the code that does this, and a *context structure* is the data structure (typically a C `struct` type) that holds instances of the saved values.

2.2 Supervisor Mode and Traps

Most machine architecture specifications include different operating modes for ordinary code (*user mode*) and the operating system kernel (*kernel mode*, sometimes also called *supervisor mode*). Code running in supervisor mode runs with increased privilege. Typically there are instructions and registers that can only be used in supervisor mode. These govern the behavior of the processor in ways that allow the kernel to run multiple ordinary ("user") processes and provide them with security guarantees. (Most control registers are restricted to supervisor mode.)

Modern machines typically have an additional mode that stands in the same relationship to supervisor mode that supervisor mode has to user mode; this is called *hypervisor mode* and allows a *hypervisor* to run multiple kernels ("guests") and provide *them* with security guarantees. This dissertation does not explicitly consider hypervisors or hypervisor mode, but the characteristics of hypervisors and the code in them are not substantially different from the characteristics of kernels and kernel code and the results will be readily transferable.

Changing between user mode and supervisor mode ("entering the kernel") cannot be done arbitrarily, or it would be impossible for the kernel to provide security guarantees. There is normally a special instruction that transfers to the kernel synchronously in order to make a system call. Execution also transfers to the kernel when an interrupt (a notification from an external device) or an exception (a failure of execution, such as accessing an invalid pointer) occurs. These events are collectively known as "traps". (On many or most architectures the distinctions among these terms are ignored; for example, on x86 one of the ways to trigger a system call is to use the `int` instruction. This stands for "interrupt".) When a trap occurs, execution continues at some known location in the kernel, which serves as an entry point. The code there is called a *trap handler* or *exception handler*; one of the things it must do is save the processor state from before the trap occurred (a context operation).

If needed (as is the case for most system calls) this allows continuing the previous execution correctly.

The way these entry points are specified varies from machine to machine. On MIPS the addresses are hardcoded into the processor (the latest architecture versions allow some, but not much, ability to control them); on RISC-V there is one that is loaded into control registers during kernel initialization; on x86 they are part of an elaborate system of in-memory tables that configure the segmented memory model.

Changing from supervisor mode to user mode is not restricted but requires a special (privileged) instruction, and in general must be closely connected to the trap handling code for kernel entry in order to restore register values.

When a hypervisor is in use most traps go to the hypervisor first, which then delegates each one transparently to the appropriate guest kernel. This is similar to the mechanism by which some traps generated by user processes can be delegated to them via Unix signals or similar mechanisms. These all involve context operations.

2.3 Virtual Memory

Most architectures intended for general-purpose use include a memory management unit (MMU) whose job is to allow virtualizing the memory spaces used by executing code. A conventional OS uses the MMU to give each user process its own address space. This isolates the processes from one another as well as from the kernel. The addresses used during execution are called *virtual addresses*; the addresses these are translated to in order to access main memory are called *physical addresses*. MMUs work in terms of a TLB (translation lookaside buffer) which is an on-chip cache of virtual to physical translations. These translations are normally expressed in terms of *pages*, which are fixed-size chunks of memory, typically either 4096 or 8192 bytes.

The TLB is a cache, and it must at times be flushed. (I shall not go into details; TLB handling is widely understood and well explained in textbooks.) It must also be filled. When an address is used and no translation is found, a *TLB miss* occurs and in cooperation with the OS either a suitable translation is loaded into the TLB or some alternate action is taken, such as killing the offending process. TLBs can be *software refill*, meaning a trap occurs for every TLB miss and software finds and loads the proper translation, or *hardware refill*, meaning the processor itself does this and generates a trap only if no translation is found. In both cases the translation is found by looking in a *page table*, a mapping structure indexed by virtual page address that contains physical page addresses and flags. Page tables are essentially always fixed-depth radix trees, where the depth is 2 for 32-bit systems and currently up to 5 for 64-bit systems.

The job of the OS's virtual memory subsystem is to maintain a page table for each user process (and sometimes also one for the kernel) such that intended access restrictions are enforced: to a first approximation, each process may only access its own memory and may not access the kernel's memory. Exceptions to this rule (such as shared memory) must be set up explicitly according to the OS's protection rules. In order to do this job the OS maintains a *forward mapping* structure for each process and a global *reverse mapping* structure, because for many operations it is necessary to know what virtual addresses map to some particular physical address. In a simple virtual memory system, the page table is the forward mapping structure. With the advent of Mach VM [24], the forward mapping structure became a higher-level system of objects and uses of those objects (this supports features like memory-mapped files) and separate from the page table. The page table serves as a cache for the higher-level structures in the forward mapping.

2.4 RAM Caches

Essentially all modern architectures include caching of main memory, due to of the discrepancy between main memory access latency and instruction execution rates. These caches usually require some level of explicit operating system control. Goldfish (my testbed OS) must exercise that control and needs to be able to synthesize the cache operations that effect it.

This section describes the conventional model of RAM caches and also the obligations placed on the operating system.

The cache is organized into layers (L1, L2, etc. where L1 is nearest the processor), each of which implements the following design independently. (Sometimes the nearer and smaller layers always contain subsets of the data in the deeper and larger layers and sometimes not; this difference is invisible to the OS and thus of interest only to hardware designers.) The L1 cache is sometimes split into separate instruction and data caches, each with its own properties. This creates additional OS obligations, discussed below.

2.4.1 Layout (“Geometry”)

The unit of memory stored in the cache is called a “line”. The size of a cache line is typically on the order of 64 bytes. (This is much smaller than virtual memory pages, which are normally 4096 bytes.) A cache has some total size (e.g. 512 KB); this is divided into slots (or “entries”) that can each hold one cache line. Most caches are *set-associative*, that is, when you look in the cache you first pick a set of entries (or just “set”) and the set includes two or more possible entries to choose from. Because the term for this is “k-way set associative”, the individual entry slots per set are called “ways”. The address of a set within the cache is called its “index”. The number of sets can be found by dividing the total number of lines by the associativity.

When the processor goes to access memory, it uses bit slices from the requested address to look in the cache. The bottom bits of the address determine the offset into the cache line and are thus irrelevant for cache lookup. The next n bits, where 2^n is the number of entries, are called the “index bits”; they are used to index into the cache and pick a set. Then in the straightforward case the rest of the address (the “tag bits”) is used to look up a “way”. Each way is *tagged* with the tag bits of the address associated with the data stored in the way (if any); these are matched in parallel by the hardware and the matching one is selected. If none matches, one of the ways is selected by an unspecified mechanism and evicted to make room.

Each cache entry may or may not contain valid data (at processor initialization time, none of the cache lines in any cache contains valid data) and that data may or may not be *dirty*, meaning that it has been modified since last written to the next layer down. Evicting a dirty cache line requires writing it back to the layer below.

Most caches are *write-back*, meaning that changes are written lazily to the next layer down. Some caches (especially on older machines) are *write-through*, meaning that any changes are written to the next layer down immediately.

Note also that modern implementations are often considerably more complex than this internally, but the OS-facing abstraction remains largely the same, as this is needed for compatibility.

2.4.2 Virtual vs. Physical

Caches may be addressed with either physical or virtual addresses. A PIPT (physically indexed, physically tagged) cache is addressed by physical address, after virtual memory translations are applied. This creates the least work for the operating system, but is not favored by hardware designers as sequencing caching after virtual memory translation increases latencies and pipeline depths. A VIVT (virtually indexed, virtually tagged) cache is addressed by virtual address. This allows cache lookup and address translation to proceed independently, but creates management obligations for the OS.

A VIPT (virtually indexed, physically tagged) cache is a hybrid: the cache *indexing* is done with the virtual address, but the *tagging* is done with the physical address. This increases the size of cache tags (instead of the leftover bits of the address being used as tag bits, one needs the whole physical page number) but decreases the management load on the OS, so VIPT caches are a common compromise.

PIVT caches are theoretically possible but are adverse in most respects and are not used in practice. VIVT caches are mostly found only on old hardware, except sometimes for instruction caches. (The consequences for instruction caches are less severe because instruction caches are read-only.)

In general, L2 and deeper caches are PIPT because the virtual memory translation delay ceases to be a problem. L1 caches are often VIPT.

Note also that for the low bits of the cache index (those that correspond to the offset into a virtual memory page) virtual and physical are the same. Sufficiently small caches are effectively always physically indexed. Unfortunately, real caches are effectively never that small. (The use of superpages, that is, virtual memory pages substantially larger than the traditional 4096 or 8192 bytes, can eliminate caching issues, but has other costs and is ultimately beyond the scope of this work.)

2.4.3 Flushing

OS obligations arise whenever accesses to memory would potentially generate inconsistent results; the OS is obligated to prevent inconsistency. This is done mostly by *flushing* cache lines or whole caches. Flush operations can *write back* dirty cache lines, *invalidate* cache lines (so the contents of a particular piece of memory are no longer stored in the cache at all), or both. Invalidating without writing back potentially loses data; however, sometimes that is correct and saves overhead, such as when releasing memory.

Flush operations can be per-line or for a whole cache at once. Lines can normally be flushed by address and often also by explicit index number and way number. (The latter is useful for iterating over a cache manually to flush all of it.) Since it is impossible to know which memory ranges are represented in the cache and which are not, and generally not worth probing, flushing operations for ranges of memory typically need to iterate over addresses one line at a time. Since each layer writes back to the one below it, in general flushing operations flush all caches, starting with the L1 cache and working downwards.

2.4.4 DMA Obligations

When a region of memory is going to be used for DMA (direct memory access) from or to hardware devices, the OS must ensure that the DMA operation sees the intended data and that the processor sees the results. Except on machines where DMA is cache-aware DMA operates on main memory. Therefore, for write operations (where the DMA device reads from main memory) the cache must be written back before the DMA begins, and for read operations (where the DMA device writes to main memory) the cache must be invalidated before the DMA begins, so that any hardware-generated write-backs during or after the DMA do not destroy the data, and also so that subsequent reads will read the new data.

2.4.5 Virtual Address Obligations

For virtually-indexed caches, if there is more than one virtual address for the same physical address, it is possible for the multiple virtual addresses to resolve to different cache indexes. If this happens, each copy will be cached independently, so unless all copies are read-only, inconsistency ensues.

There are two ways to work around this problem: making all the mappings uncached, which reduces performance, or (for VIPT caches only) ensuring that all mappings resolve to the same cache set; the tags will then match and all will be well. For this to happen the index bits of the virtual address, excluding those that fall within a virtual memory page, must be the same. Each value for these bits is known as a "color", and if the OS can take cache colors into account when allocating virtual addresses it can avoid needing to use uncached mappings. To do this the machine-independent code must have a suitable allocator and it must be able to discover from machine-dependent code how many cache colors there are.

If the cache is virtually tagged, multiple mappings *must* be uncached or read-only. However, another problem arises with virtually tagged caches: switching from one process to another changes the page translations and consequently it is possible to read another process's data out of the cache unless the entire cache is flushed at every context switch. (Context switch here meaning switching from one user-level process to another and perhaps also from user-level processes into and out of the kernel, depending on other properties of the machine.) This limits the consequences of the restriction on multiple mappings (because it is rare for one process to map the same memory for concurrent access at multiple addresses) but introduces substantial overheads. On some machines with virtually tagged caches the cache tags also include the address-space ID used by the MMU, in which case the cache only needs to be flushed when recycling address space IDs. On these machines multiple mappings across processes must still be uncached, however.

For certain kinds of multiple mappings (those done by the kernel for kernel purposes, such as zeroing pages) it is possible to flush the cache instead of bypassing it. For example, when copying pages due to `fork` the kernel must map the pages in its own address space (or use an existing in-kernel mapping); it is sufficient to flush the user virtual addresses before copying and flush the kernel addresses after copying. This also applies to page zeroing, mapping pages for swap I/O, and so forth.

It is also important with virtually indexed caches to make sure that multiple mappings do not accidentally accumulate over time: whenever a page is released/freed, its virtual addresses should be invalidated. (Otherwise when the underlying physical page is allocated again, there might be stale dirty cache entries that lead to memory corruption.)

2.4.6 Instruction Cache Obligations

When the L1 cache is split into separate instruction and data caches, writes to the data cache are not necessarily reflected in the instruction cache. On such machines it is necessary to write back the data cache and invalidate the instruction cache after changing executable code. This applies to various things kernels sometimes do (such as copying trap handling code to well-known addresses) and also to just-in-time compilation. It also potentially includes when paging in code or loading executables. Since this involves freshly allocated pages, for VIPT and VIVT data caches the obligations above will mean that the pages are clean in the instruction cache; however, depending on how the I/O is done it may still be necessary to write back the data cache. For PIPT data caches, however, this is not true and in general the instruction cache also needs to be invalidated.

2.5 The Operating Systems in This Dissertation

This dissertation mentions a number of operating systems, some in passing. These are summarized herein for reference.

2.5.1 OS/161

OS/161 [14] is a teaching OS first written in 2001. It is small compared to a “real” OS, but large compared to the typical teaching OS. It is distributed to students lacking major functionality, which the students then implement. For the purposes of this dissertation, I use “OS/161” to mean this skeleton along with the solution set versions of the student-written code, most notably the virtual memory system. The current release (2.0.3) contains about 51000 lines of code (counted with `wc -l`, excluding blank lines and comments) of which about 31000 are in the kernel and the rest are user utilities, tests, and benchmarks.

OS/161 is designed to provide a realistic kernel development experience for students while glossing over some of the worst features of real kernel development on real hardware. Natively it runs on its own platform, System/161, which is a MIPS emulator designed specifically to support rapid kernel development: it has reasonably realistic but simplified hardware devices and a simple (though multiprocessor) mainboard architecture. Its primary limitations from a realism standpoint are that it does not model caches at all (either for timing or consistency management) and that it has a simplistic instruction timing model with a hardwired clock rate characteristic of the early 1990s. Timing measurements on System/161 are consequently mostly of value only for comparing to other timing measurements made on System/161 (such as performance contests among students) and cannot, except in the broadest terms, be related to performance on real modern hardware. This is sufficient for the purposes of the type of work in this dissertation, however, where the primary goal is correctness and any performance criteria are limited to showing that performance is not made shockingly worse. (For synthesis, even being able to produce code that works but is horribly slow is a victory.)

While OS/161 is something of a toy, it is (concomitantly with being larger) more “real” than one might expect for a teaching OS. It qualifies as a real OS in the traditional sense of being able to run Rogue. It has successfully been ported to at least one real hardware platform (the Raspberry Pi). It also does not support a number of standard but expensive features in production OSes, such as support for memory-mapped files, unification of the file system buffer cache and the virtual memory system, shared libraries, or threads in user programs. That said, the version with the solution code is reasonably functionally complete in a basic way.

2.5.2 Goldfish

Goldfish is a refinement of OS/161 2.0.3 (with the solution code) for the specific purpose of supporting the work in this dissertation. Most notably, I took the OS/161 virtual memory system (which was never intended to be portable, as it is solution code and thus should be at most a polished version of something a student might reasonably write), removed its dependence on MIPS hardware properties, and made it machine-independent with a small machine-dependent backend. I have not made other major structural changes, but I have rearranged some of the machine-dependent interfaces. All of this is discussed in more detail in Chapter 8.

2.5.3 NetBSD

NetBSD is a production OS descended from 4.4BSD. It is functionally complete and runs natively on a wide range of hardware. (In fact, it is to some extent notorious for maintaining support for necrocomputing platforms, with the explicitly stated rationale that they help keep the main machine-independent code base honest.) It is for various reasons often treated as the reference implementation of a Unix-style OS, and is thus used for contrast to OS/161 and Goldfish in the design discussions in Chapter 8.

2.5.4 FreeBSD

FreeBSD is another production OS descended from 4.4BSD. It is also functionally complete and runs natively on a wide range of hardware. (At one time it was not intended to be portable but rather specialized for x86, but that time was a long time ago now.) FreeBSD and NetBSD diverged in the early 1990s and are now, while similar in broad outline, often quite different in detail, which often makes FreeBSD an interesting design contrast to NetBSD.

2.5.5 Linux

Linux is a clone of traditional Unix rather than a descendant; it is thus quite different inside. For the purposes of this dissertation it has a significant drawback, which is that its machine-dependent interfaces are much broader than NetBSD's or FreeBSDs. Many things that are machine-independent in the BSDs are machine-dependent in Linux, such as error numbers and system call numbers. (This was done for a reason – compatibility with the native OS when doing ports to new machines – but makes it a poor choice for work related to portability and machine independence.)

2.6 The Machines in This Dissertation

This dissertation also mentions a number of machine architectures. These are now described briefly in the following subsections.

2.6.1 MIPS

MIPS is a RISC architecture (one of the original RISC architectures, in fact) with both 32-bit and 64-bit versions. (There are, in fact, many versions, perhaps more than other architectures even of a similar age.) The version used in this dissertation is the MIPS software implementation found in System/161, the native environment of OS/161. This is a mixture of MIPS32 and MIPS-I, with a few custom changes. MIPS-I is the original version from the late 1980s; MIPS32 dates from the early 2000s. The choice of versions was dictated by specification availability.

System/161 uses the simpler MIPS-I MMU (but not the horrifying MIPS-I cache) for pedagogical reasons. It offers a fully software controlled (thus, software-refilled) fully-associative and fixed-size TLB, which gives a great deal of flexibility to operating systems to implement virtual memory as they see fit. (Later MIPS TLBs are still software-refill, but require pages to be mapped in pairs and are substantially more complex to work with, which we felt was unhelpful for students.) The "native" page table arrangment is in practice a two-layer radix tree (like essentially all modern or semi-modern 32-bit machines) but where the table itself (the lower layer) is accessed linearly by virtue of being mapped into the kernel's memory using the upper layer itself as a page table. However, arbitrary other layouts, including the conventional radix tree, are possible. The memory space is divided into hardwired[1] segments, such that user programs are allowed to use the bottom half of the address space, the kernel is mapped into the top half of every address space, and part of the kernel's space is direct-mapped to physical addresses so as not to require TLB handling or consume TLB entries.

There are 32 general-purpose registers, one of which is wired to be always zero. Instructions are mostly three-operand. The chief other interesting architecture feature is that, like many early RISC designs, there are exposed pipeline effects, specifically branch delay slots. (That is, after a branch instruction, the processor executes one more instruction before actually branching.) Because MIPS assemblers have a mode that hides branch delay slots and other pipeline effects (by introducing nops as needed) we do not, for the time being, handle them in our synthesis engine. Taking advantage of the optimization opportunities offered by manipulating branch delay slots directly is a topic for future work.

2.6.2 RISC-V

RISC-V [25] is a recently developed open machine architecture. It has 32-bit, 64-bit, and (experimental) 128-bit variants. There is also a cut-down 32-bit embedded version ("RV32E"). This dissertation uses the 32-bit variant ("RV32I"). (In RISC-V terminology, it actually uses the 32-bit variant with standard extensions, "RV32G".) I would not expect (modulo possible 64-bit cleanliness bugs in tools) the 64-bit variant to behave materially differently with respect to the material herein. There are 32 general-purpose registers, one of which is wired to be always zero. Instructions are three-operand. There are no delay slots.

The MMU has a hardware-refill TLB of unspecified geometry. It uses a 2-layer page table radix tree on 32-bit systems and either 3 or 4 layers on 64-bit systems. The design assumption is that the kernel is mapped into the top of every address space. A protection mode is provided to help prevent accidental accesses to user memory when executing in the kernel.

2.6.3 ARM

ARM is a RISC architecture (at least nominally) with 32-bit ("AArch32") and 64-bit ("AArch64") variants as well as a large number of versions. (There are not as many major versions (8) as MIPS (17), but the versions differ from one another substantially more.) There are 16 general-purpose registers,

[1] Recent versions make the segmentation somewhat programmable.

one of which is the program counter. Instructions are three-operand and often four-operand. There are no delay slots.

2.6.4 SPARC

SPARC is a RISC architecture (also one of the original RISC architectures), now effectively dead. It was originally 32-bit but only the 64-bit version remained in use past the 1990s. It is notable for a feature known as *register windows*, a scheme that provides a stack of register sets in hardware to match the stack frames of procedure calls. There are 32 registers (including the ones manipulated by the windows) and instructions are three-operand.

2.6.5 x86

x86 as used here refers to Intel's widely used 80x86 instruction set, either the original 32-bit version (80386 and up) or the 64-bit version (sometimes called X64) developed by AMD. (Intel's own earlier 64-bit architecture, IA64, is entirely different.) The 32-bit version has only 8 general-purpose registers; the 64-bit version extends this to 16. Instructions are two-operand. There are no delay slots.

The 32-bit MMU is extraordinarily complicated, having both a segment-based system and a page-based system layered one on top of the other. A detailed discussion is beyond the scope of this section. The TLB is hardware-refill, and the page tables are a two-layer radix tree. The kernel is meant to be mapped into the top of every address space, though the Meltdown attack [19] made this dangerous and complicated steps are necessary for mitigation. The 64-bit MMU simplifies away the segments and increases the page table depth to up to 5.

While we have experimented with assembly synthesis for x86, I have not attempted to apply the tools in this dissertation or generate a Goldfish port. However, x86 is mentioned frequently in the text to follow for comparative and analytic purposes and the tools have been designed with possible x86 support in mind.

Chapter 3

Program Synthesis and Verification Background

This chapter provides a brief background on program synthesis: what it does and doesn't do, how programs to be synthesized are specified, and how it relates to program verification, compilation, and superoptimization. It also briefly introduces our project's assembly language synthesis engine (Capstan) and how it fits into this picture. It does not discuss *how* synthesis works, except in the broadest outline, as the goal of this dissertation is to apply our state-of-the-art program synthesis to a real-world problem (porting of operating systems), not define or extend the state of that art.

3.1 Synthesis and Specifications

Program synthesis refers to creating an executable program, or executable code, from a typically non-executable specification. There are various possible ways to do this. Much modern work, including ours, uses a satisfiability-modulo-theories (SMT) solver [7, 6] and a technique called CEGIS (counterexample-guided inductive synthesis) [29]. CEGIS is an iterative method that converts the problem

$$\exists program, \forall input, MatchesSpec(input, program(input))$$

into a purely existential search that an SMT solver can handle reasonably effectively.

In some synthesis work [9, 22] the specification is a set of input and output examples; in others, including ours, the specification is expressed in Hoare logic as a precondition and a postcondition, and *MatchesSpec* takes the form

$$pre(input) \rightarrow post(program(input))$$

In the case of assembly language synthesis, *input* and *output* are machine states, including potentially both registers and memory. Given an input state that satisfies the precondition, running the synthesized program should yield a machine state that satisfies the postcondition.

The synthesis engine encodes the space of programs, the space of inputs, the execution semantics of the programming language, and the specification as a sequence of (large) formulas in first-order logic. The CEGIS method alternates between "guess" and "check" steps, in which a "guess" step uses the solver to produce a candidate program and the "check" step uses the solver to look for an input state on which the program does the wrong thing. On success, this yields a program that meets the specification. (In our case, because we do not allow loops, success always yields a correct program. Synthesis for Turing-complete languages gets more complicated.) The same encoding can be used to verify an already existing program, handwritten or generated some other way; we will see this applied in Chapter 6.

Synthesis begins by trying to synthesize a single-instruction program. If that fails, it moves on to a two-instruction program, then three, and similarly until either the solver runs out of memory or fails to terminate in a reasonable length of time. Thus, it will always pick the shortest output program, at least among the programs it is capable of generating. Note however that more instructions have an exponentially larger search space of possible programs and take longer. Our synthesis engine scales up to output programs of about five instructions [16], though the exact number depends on the specification. This is not very many; assembly language synthesis is difficult [15]. Chapter 6 discusses avoiding this limit with a special-case code generator for context operations. Chapter 7 discusses working around this limit by explicitly composing blocks. Other work from our group uses deduction rules [21] within the synthesis engine to raise the limit (for certain classes of specification) by automatically decomposing the specification.

3.2 Synthesis and Verification

As hinted in the previous section, the synthesis engine can also verify programs. Verification, indeed fully automated verification, is inherent in synthesis: not only does the CEGIS method rely on verification (the "check") step to automatically produce counterexamples to proposed programs, but in general any synthesis engine that does not verify its output is free to, and in general will, simply produce rubbish.

3.3 Synthesis vs. Compilation

One interesting question is where, among the possible strategies for program generation, the line between "synthesis" and "compilation" lies. My view on this is that compilation is a transformation from one executable representation to another, and synthesis is a search in the space of executable representations based on criteria that do not directly express the execution to be performed. But the distinction can be subtle and is not always clear: for example, register allocation, which is a highly traditional stage of compilation, is a search. Meanwhile, deductive synthesis [21] is perhaps not really a search.

Note also that this definition includes some problems that are not traditionally thought of as synthesis, most notably query planning in relational databases.

For the purposes of this dissertation, however, this question is immaterial; one can think of a spectrum of ways to specify blocks, that result in compilation at one end and synthesis at the other. In large part my goal is to allow using a range of tools that fall at various points of the spectrum to generate blocks that can be composed effectively.

Note that in this dissertation I use the term "generate" to cover code generation by compilation, synthesis, or points between.

3.4 Synthesis vs. Superoptimization

Superoptimization [20] is closely related to synthesis, especially assembly synthesis. In both cases the goal is to do a broad search to find a sequence of instructions meeting a specification. In the case of superoptimization, the specification takes the form of equivalence to an existing code block. This means that a superoptimizer never fails: in the worst case it can always return the original code. It also allows a wide range of implementation tactics that do not work for synthesis, such as automatically subdividing the problem by computing intermediate machine states.

Chapter 4

Problem Analysis

The first step in a successful project is to gather understanding of the problem, and the first step of that is to identify what must be done, and, equally importantly, what need not be done. Given a set of high-level project goals, analysis of the problem space yields lists of more specific requirements that can be used to propose system designs. That said, requirements analysis up front has inherent limits and usually in the course of a research project additional understanding leads to additional or updated requirements. The following discussion includes both.

4.1 Initial Requirements Analysis

Recall from Chapter 1 that the overall goal was to reduce the cost of porting operating systems by generating the required code. Accordingly, the first step in requirements analysis was to take an inventory of the code that needed to be generated, to help decide how best to go about generating it. I went through two operating systems (OS/161 and NetBSD) and examined all their machine-dependent code in detail, then wrote a report about what I found, concentrating on the nature of the code, what it did, and how one might go about generating it. Note that I did not do line counts, or divide up the code in detail by category or topic – at the time, I assumed that would come later. The report extended to 18 pages; I am not going to reproduce it (or even summarize it), just report on the conclusions we and I drew from it.

Machine-dependent code in operating systems (the ones I examined, but not only those) is generally a mix of C and assembly language. Most, though often not quite all, of the assembly code is written in assembly because it must be: it does things that cannot be expressed in C or other high-level language. The most common reasons for this are: inspecting or updating machine state that compilers do not know about (e.g. control registers, caches); manipulating higher-level constructs in ways that compilers do not permit (e.g. switching between stacks); and retaining precise control of all machine state to avoid corrupting or losing information that must be preserved.

The first and most fundamental requirement was that we needed to be able to generate such assembly code. One approach (perhaps the most conventional in the systems world) would be to write a custom code generator for each distinct code generation problem. This would be a large engineering effort, with a substantial risk of failure, in the sense of not actually making ports less expensive. Another approach would be to design a domain-specific language capable of expressing the needed code in a machine-independent way, then write a compiler for it. This would also have a substantial risk of failure, in that designing such a language is a potentially intractable problem – certainly it is not an easy problem or it would have been solved already. These approaches are not exactly different so much as opposite ends of a scale, so various intermediate points would also be possible. (It is actually two related scales, one of language generality – in the first case, there is one trivial language per problem – and one based on how much of the hard material is open-coded into the compilers vs. how much is expressed in the DSLs.) However, we chose to embrace program synthesis instead.

The second requirement is that we needed to be able to generate such code across a range of machine architectures. Parameterizing compilers with machine descriptions has an extremely long history[1] but program synthesis has different requirements, and we anticipated that trying to use off-the-shelf descriptions would be problematic. So we also needed a machine description language suitable for synthesis.

Program synthesis has several advantages in our context. Most importantly, we only need to be able to write a specification for the code we need, rather than an algorithmic expression of it. This is extremely helpful when the goal (such as disabling interrupts) is machine-independent but the mechanism and procedure for doing so is not. We will see that for much of the code we need to generate the specifications are comparatively straightforward. Synthesis also allows separating the problem of machine-independence from the problem of code generation: if we can write a

[1] Blindell's summary book on instruction selection [3] observes "...many such languages and related tools have appeared–and then disappeared–over the years" (p. 14) and cites a survey paper from 1969.

machine-*dependent* specification of the code we need, we can generate that code. Therefore, we only need a mechanism for taking a machine-independent specification and instantiating it as (or lowering it to) a corresponding machine-dependent specification for a given hardware target. We do *not* need to work directly from machine-independent specifications or try to formulate a language of machine-independent actions.

Synthesis has two major drawbacks, however. One is that its scalability is limited. It can generate only comparatively small blocks of code of only a few instructions at a time. Even at the beginning it was clear that we would need some kind of ability to compose these blocks into larger units, since some of the assembly procedures we found were a hundred instructions or more. The other is that it can be very slow and compute- or memory-intensive; even for small blocks the resource requirements can turn out to be prohibitive. Excessively complicated specifications increase the risk of synthesis failure.

Thus we require the following:

- A language for writing machine-independent specifications.
- A way to lower machine-independent specifications to machine-dependent specifications.
- A way to synthesize assembly language blocks from machine-dependent specifications.
- A machine description language suitable for assembly synthesis.
- A machine description language suitable for lowering specifications.
- A way to compose synthesized assembly language blocks.

Each of these requirements is met by one or more of the tools discussed in the next chapter.

A “block” is a sequence of a small number of machine instructions in assembly language, akin to a basic block in a compiler, but often smaller. These will be the units of code generation. We explicitly allow blocks to contain internal branching, unlike compiler basic blocks.

I also concluded that if there were categories of blocks we could readily generate via other techniques, we should do so, and the tool or tools for block composition should take this into account.

Note that while much machine-dependent code is written in C, not assembly, we did not consider synthesis of C code (or anything similar to C code) a requirement, concluding that it was better not to multiply entities and that we could use assembler for everything. In retrospect this was likely a mistake, and this would probably be a good avenue for future work.

4.2 Initial Non-Requirements

As noted in Chapter 1 we specifically excluded some things from the project scope. For the purposes of this dissertation, I assume that someone else has done the work to generate code in these categories. (And where necessary to get my test environment running, I have written the code by hand or cribbed it from existing ports of existing operating systems. See Section 8.6.)

Initially, we excluded just the compiler, on the grounds that the compiler is both a large project in its own right and also a different kind of code. (We later interpreted this to include other compilers besides the main system compiler; for example, NetBSD includes a just-in-time compiler for packet filters.) We (I) actually did some work on retargeting the compiler tools (primarily the assembler and linker); that was an earlier separate phase of the project and is not covered in this dissertation. For the purposes of this dissertation the compiler tools are excluded, as are other linker-like things, such as the ELF dynamic linker and the kernel module loader. We decided that debuggers also belonged to this category, both the system debugger and the custom in-kernel debugger that some OSes (including NetBSD) have.

We quickly also concluded that while it would be great to synthesize a bootloader, bootloaders for some platforms are highly complex in their own right [10].

Meanwhile, although device drivers are a worthy target, they are also a different kind of code, and the Termite project [27, 26] had been working on the problem for some years already. We were skeptical of being able to make any additional progress in our project's time frame and concluded it would be better to adopt their work rather than reinvent it, and concentrate on other things.

Somewhat later we excluded code related to handling system boards, because collaborators [1, 17] had taken up that problem. Specifically (drawing from the inventory) this covers:

- Code that probes for memory, memory types, and additional CPUs/cores, plus any NUMA geometry these might have.

- Code that probes for buses and devices attached to buses.

- Code for routing interrupts, allocating device memory, and handling other bus configuration issues.

- Code for talking to devices over buses.
- Code for interacting with system firmware.
- Definitions for constants associated with system boards and SoCs, most notably a limit for the number of CPUs/cores.
- Installer widgetry for e.g. knowing which devices to try to install onto.

4.3 Experience

As mentioned above, it was clear from the very beginning that we would need some way to compose the assembler blocks output by the synthesis engine. At first it was not clear that this really amounted to anything more than concatenation; after all, given block *A* and block *B*, as long as the postcondition of *A* implies the precondition of *B*, then the composed block is just *AB*; its precondition is the precondition of *A* and its postcondition is the postcondition of *B*. Consequently to begin with this requirement was conflated with another that appeared early on.

That requirement in turn was the need to produce stylized output. One of the things the inventory revealed was that a substantial amount of the material in an operating system port is not code, but declarations and definitions. These include base types (e.g. which C integer type should be used as `size_t`), some structure types (e.g. the structure registers are saved into when an interrupt happens), many kinds of constants (ranging from integer limits associated with integer types to magic address ranges in the virtual memory layout), and also, often, preprocessor macros defining properties of the platform to be used by existing machine-independent code. Furthermore, every OS requires various auxiliary files as well; for example, in almost all cases kernels linked with GNU `ld` need a linker script. There are also makefile fragments, kernel configuration data, and other such things.

Once I started digging into these and trying to generate them, it quickly became clear that we were going to need some kind of templating facility to produce these files. It was only much later that it became clear that this and block composition were fundamentally different problems. (See below.)

It also became clear that we were going to need a way to generate the things that the templater would insert into its templates. It is reasonable to treat basic things such as the definition of `size_t` as fundamental properties of the machine, and just include the proper type as a fact about the machine in the machine description. That requires a way to name the fact needed, and perhaps a tool to read the machine description and extract it. Other more complicated things (such as the register save structure) require a way to specify and generate them, perhaps by synthesis.

Our synthesis engine encountered a number of teething problems during its early development, and it became clear that we should go ahead with more traditional code generation methods anywhere it seemed reasonably feasible. One of the results of the code inventory was that there is a lot of *context handling* code in operating systems. Naively one might think that this would be limited to, e.g., switching processes within the kernel, and saving registers on kernel entry and exit. However, it turns out that there are more than that; for example, the standard C functions `setjmp` and `longjmp` are context operations. The `ptrace` system call is responsible for allowing a debugger to inspect a debuggee's execution context. When a program crashes, producing a core dump requires saving out the register values it was executing with. Meanwhile it turns out that saving and restoring registers is a particularly adverse problem for program synthesis. (This is discussed in detail in Chapter 6.)

For these reasons we decided that we wanted a *context-switch compiler*, that would accept some kind of specification of the registers to save and restore and then generate code to do it using traditional compilation methods. This in turn immediately complicated the composition problem, because now some assembler blocks are generated from pre- and postcondition pairs and some from other specifications, and composing them safely is no longer trivial.

It also became clear that we could carve off an additional category of assembler blocks: calling sequences. While some of the assembler code found in the inventory was more or less standalone, in the sense of not calling anything and, while callable, taking no arguments, most of it interacts with the rest of the kernel to a greater or lesser extent: it makes calls to C code and is usually called from and takes arguments from C code. We need to be able to generate blocks that place arguments and make calls, and also the prologue/epilogue blocks that set up the stack and know where arguments are. Since calling conventions are complicated, avoiding exposing the synthesis engine to their specifications seemed like a win. So we decided to have a separate tool for these too, which complicates the composition problem more: the calling conventions (and thus the call generator tool) decide where arguments are placed, which means that while most blocks consume specifications, call, prologue, and epilogue blocks *produce* specifications.

Meanwhile some preliminary investigation into synthesizing synchronization code (spinlocks, memory barriers, an atomic operations library) had concluded that it was, at least for the time being (see Section 9.2), not worthwhile. To see why, consider the specification for a memory barrier instruction, the specification for an abstract memory barrier operation required by the OS, and the machinery required to be able to write these specifications down and feed them to a solver for synthesis. Consider also that running the solver on this might be expensive. Then consider that the output is, in all cases, one instruction, and in most cases, one of only two or three different kinds of memory barriers supported by the machine. Synthesizing this code has a truly awful power-to-weight ratio, and it makes far more sense to just have the porter specify which instruction to use directly. This applies to all the concurrency code (which for proof-of-concept only needs to include memory barriers and a spinlock implementation, which though larger than a memory barrier is still typically only a few instructions) and also to some other simple machine-level actions that are otherwise difficult to specify usefully, such as idling the processor. This also further complicates composition, in that some blocks will be handwritten, and their specifications may not be fully expressible in the general-purpose specification language.

These complications regarding composition were in direct conflict with the desire to have the templating tool be simple; after all, inserting values into template strings is far from a research problem, and spending time on it is not expedient. Eventually this forced the realization that the templater and the composition tool needed to be different things.

Finally, implementing the templating tool revealed that we needed a common interchange language for types and values. It followed that this language could also serve some machine description purposes and take the place of a tool for extracting basic definitions (and handwritten code blocks) from a more complicated machine description.

Each of these additional requirements is also met by one or more of the tools described in the following chapter.

Chapter 5

Aquarium of Languages and Tools

To satisfy the requirements from Chapter 4, I designed an entire ecosystem of languages and tools. It is a complex environment with many interacting elements, some of which are easily confused, so I shall describe it as a whole from several different perspectives before going on to discuss the individual pieces. I will also provide some background on the various names as this should help the reader follow them.

First, here is the list of names and their core functions:

- Alewife (specification language)
- Anchor (Alewife specification processor)
- Cassiopea (machine description language)
- Capstan (synthesis engine)
- Grayling (block generator)
- Grouper (compositional code generation tool)
- Goldfish (operating system)
- Tuna (data interchange language)
- Penguin (experimental MIPS synthesis engine)
- Nautilus (block generator)
- Hermitcrab (templater)

5.1 Synthesis Core

Our core synthesis languages and tools are as follows:

- *Alewife* is a language for writing machine-independent block specifications as pre- and post-conditions. It is based on predicate refinement. We chose the name because it is both a fish and a district and subway stop near the university.

- *Anchor* is the compiler for Alewife: it takes a machine-independent specification written in Alewife, and using machine-dependent *lowering* files to instantiate Alewife predicates, produces a machine-dependent specification (in a dialect of Cassiopea) that can be used for synthesis. The lowering files are written using Cassiopea expressions as well. The name arises because its job is *lowering* specifications, that is, converting them to a lower-level representation. (There has been some question since as to whether "lowering" is the correct terminology, but so far the names remain.)

- *Cassiopea* is a register-transfer-style machine description language. It is used to specify the semantics of machine instructions. It is executable but not Turing-complete. It is named for a jelly(fish) that features symbiotic photosynthetic algae, which is for some reason spelled without the 'i' that you, I, and everyone else expect to see based on the names of the constellation and legendary queen.

- *Capstan* is our assembly synthesis engine, the tool associated with the Cassiopea language. It is retargetable via Cassiopea machine descriptions. It accepts machine-dependent block specifications (written with Cassiopea expressions, output by Anchor) and produces blocks of assembly code. It can also verify preexisting blocks of assembly code, or execute them against a specific initial machine state. The name arises because it works in concert with Anchor.

These languages and tools are not topics of this dissertation, and though I have contributed substantially to them they are mostly not my work. However, I make heavy use of them: all the synthesis is done with Capstan and the machine descriptions for it are written in Cassiopea. Alewife and Anchor are used less, because Grouper's front end serves the same purpose (see Chapter 7) but the specifications for one set of synthesis problems do not use Grouper and are written in Alewife instead. See Section 8.3.9.

The language specifications for Alewife and Cassiopea may be found in a technical report [13], and we have a paper under submission [16].

There is an additional minor file format: in many contexts, code blocks are represented as sequences of instructions written as S-expressions. When these stand alone they are (somewhat confusingly) referred to as `.prog` files, even though they essentially never contain whole programs. We introduced this format for Capstan in order to avoid becoming entangled in parsing assembly language syntax. Despite various commonalities, assembly language syntax varies substantially from machine to machine.

Note that Alewife and Cassiopea are specifically designed to work together.

5.2 Big Fish

The major fish in the Aquarium are as follows.

- *Grayling* is a special-purpose tool for handling context operations. It generates register save and restore blocks as well as the context structure types these blocks work with. It is a cold-water fish[1] and thus operates in a comparatively hostile environment: context handling code is particularly difficult to synthesize. Grayling is my work, a topic of this dissertation, and discussed in Chapter 6.

- *Grouper* is a tool for composing code blocks generated with other tools (or provided in canned form in Tuna) into whole procedures. The name reflects what it does, which is also conveniently a kind of fish. Grouper is my work, a topic of this dissertation, and discussed in Chapter 7.

- *Goldfish* is an experimental operating system, derived from OS/161 [14], and a testbed for using the Aquarium to generate operating system ports. The name arises because it is a small, friendly operating system. Goldfish is my work, a topic of this dissertation, and discussed in Chapter 8.

Grouper's specification language does not have its own name; if it did, it would be another entry, much like the languages in the synthesis core. The full abstract syntax is given in Appendix B. (The typing rules and expression semantics are not given as they are standard and not interesting.)

[1] I thought it arctic when I chose the name, but I was wrong. Wikipedia says even the arctic grayling (*Thymallus arcticus*) is found in the continental United States.

5.3 Small Fry

There are also other less important fish in the Aquarium.

- *Tuna* is a language that provides a common interchange format for types and values. It has a limited expression evaluation ability to reduce the need for cut-and-paste in handwritten material, but should not be thought of as executable. The name refers to its original envisioned role: a way to write down "canned" machine description material. Tuna is my work and is an incidental part of this dissertation. The language specification may be found among the code artefacts, in the Grouper source tree. A brief discussion appears in Section 5.5.

- *Penguin* is an assembly synthesis engine specific to the MIPS architecture. I designed and built it to allow experimentation with assembly synthesis without the additional complications associated with machine descriptions and retargetability. The name arises as follows: I had been using birds as project code names for some years, and it started out separate from the rest of the project; so it looks like a fish but is really a bird. Penguin is not a topic of this dissertation, nor is it used in the results, so while I have included it in the ecosystem description for completeness it will not be mentioned again after this chapter.

- *Nautilus* generates code blocks that interact with C calling conventions. This means both calls to C code and also for the prologue and epilogue linkage blocks needed in assembly routines that are themselves called by C code. It was conceived to isolate the rest of the project from the complexities of calling conventions and the name arises because at the time we considered what it does boring. Nautilus is my fault; it is a hack and does not merit more discussion in this dissertation than the minimal account found in Section 5.6.

- *Hermitcrab* is a text templating tool whose job is to insert material generated by other tools into output files of arbitrary and perhaps stylized formatting. (For example, it can produce C header files containing context types generated by Grayling.) Hermit crabs insert themselves into the shells of arbitrary other molluscs. Hermitcrab is important for the generation of Goldfish ports but it is not itself a research artefact. Hermitcrab is my work, an incidental part of this dissertation, and described in Section 5.7.

Many of the tools have their own input formats as well; these are discussed along with the tools.

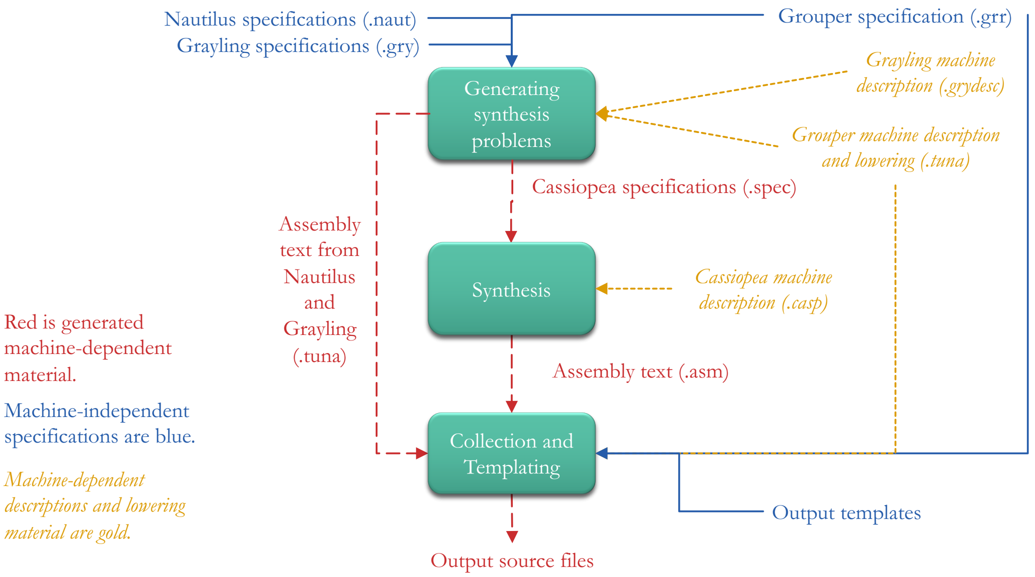
Nautilus specifications (.naut)
Grayling specifications (.gry)
Grouper specification (.grr)
Grayling machine description (.grydesc)
Grouper machine description and lowering (.tuna)
Generating synthesis problems
Cassiopea specifications (.spec)
Assembly text from Nautilus and Grayling (.tuna)
Synthesis
Cassiopea machine description (.casp)
Red is generated machine-dependent material.
Machine-independent specifications are blue.
Machine-dependent descriptions and lowering material are gold.
Assembly text (.asm)
Collection and Templating
Output templates
Output source files

Figure 5.1: *Top level Aquarium interaction diagram. The shaded green boxes are stages of the port generation process.*

5.4 Interactions

Having introduced all the tools, I can now discuss how they interoperate.

Figure 5.1 shows the top-level flow diagram for Aquarium code generation: first we generate synthesis problems with Grouper (this involves also generating the compiled blocks), then we synthesize, then we collect the output and insert into templates to produce output source files. The shaded green boxes represent sub-diagrams, stages of the process, not individual steps or tools. Machine-independent specifications are shown as blue solid lines; generated machine-dependent files are shown as red lines with large dashes; and machine-dependent files written by the porter are shown as gold lines with small dashes.

The top stage is expanded in Figure 5.2. We generate synthesis problems first by running Nautilus (and the prepare phase of Grouper); then running Grayling; then running the Grouper generate phase. This produces assembly text for compiled blocks, synthesis problem specifications in Cassiopea, and various types and constants from Grayling.

Figure 5.3 expands the Nautilus stage. The purple boxes are tools. The arrow between Nautilus and Grouper prepare also consists of block specifications.

Figure 5.4 expands the Grayling stage. In addition to producing output assembly, block specifi-

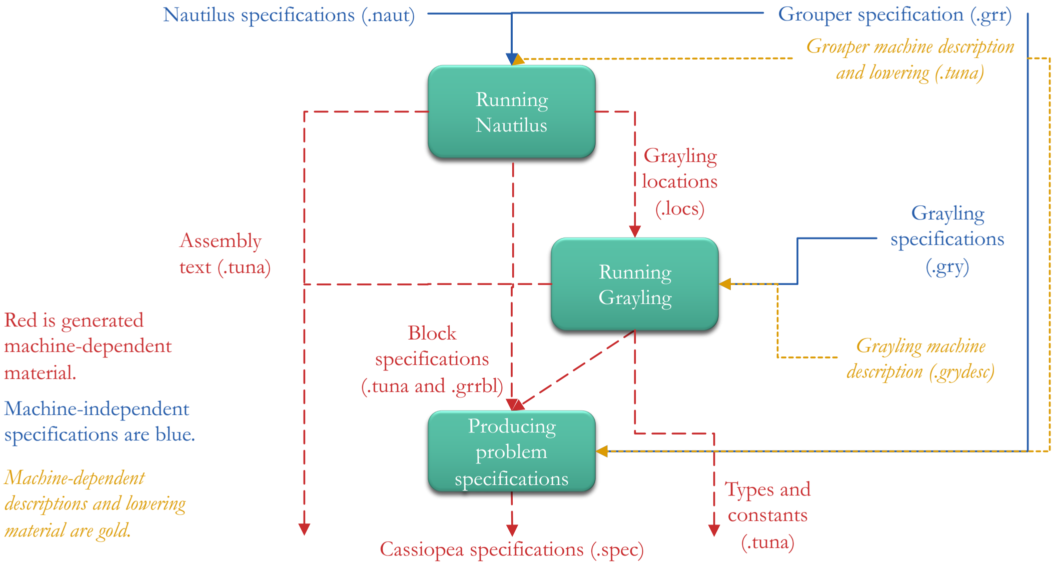
Nautilus specifications (.naut)
Grouper specification (.grr)
Grouper machine description and lowering (.tuna)
Running Nautilus
Grayling locations (.locs)
Assembly text (.tuna)
Grayling specifications (.gry)
Running Grayling
Red is generated machine-dependent material.
Block specifications (.tuna and .grrbl)
Grayling machine description (.grydesc)
Machine-independent specifications are blue.
Producing problem specifications
Machine-dependent descriptions and lowering material are gold.
Types and constants (.tuna)
Cassiopea specifications (.spec)

Figure 5.2: *Expansion of the specification generation stage into three substages.*

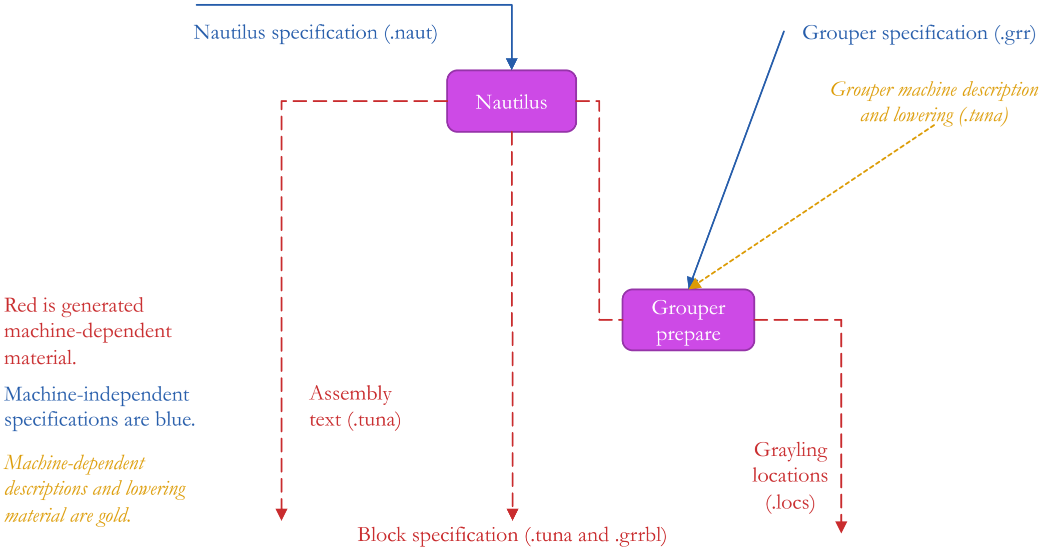
Nautilus specification (.naut)
Grouper specification (.grr)
Nautilus
Grouper machine description and lowering (.tuna)
Grouper prepare
Red is generated machine-dependent material.
Machine-independent specifications are blue.
Assembly text (.tuna)
Grayling locations (.locs)
Machine-dependent descriptions and lowering material are gold.
Block specification (.tuna and .grrbl)

Figure 5.3: *Nautilus and Grouper prepare flow diagram; expansion of the "Running Nautilus" stage. The purple boxes are tools. The line between Nautilus and Grouper prepare (the prepare phase of Grouper) is also block specifications.*

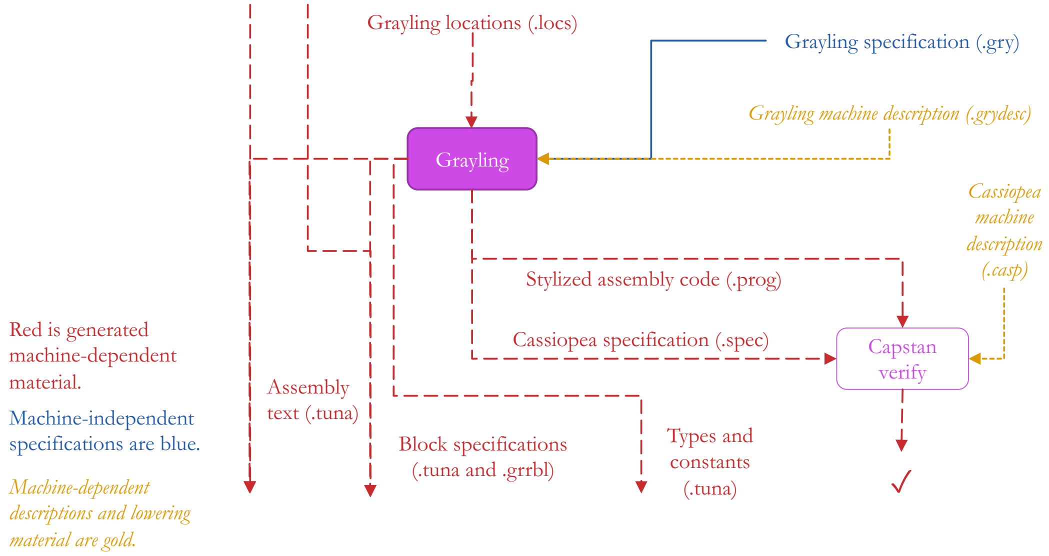
Grayling locations (.locs)
Grayling specification (.gry)
Grayling machine description (.grydesc)
Grayling
Cassiopea machine description (.casp)
Stylized assembly code (.prog)
Cassiopea specification (.spec)
Capstan verify
Red is generated machine-dependent material.
Machine-independent specifications are blue.
Machine-dependent descriptions and lowering material are gold.
Assembly text (.tuna)
Block specifications (.tuna and .grrbl)
Types and constants (.tuna)
✓

Figure 5.4: *Grayling flow diagram; expansion of the "Running Grayling" stage. The Capstan box is not filled because Capstan is not itself a topic of this dissertation.*

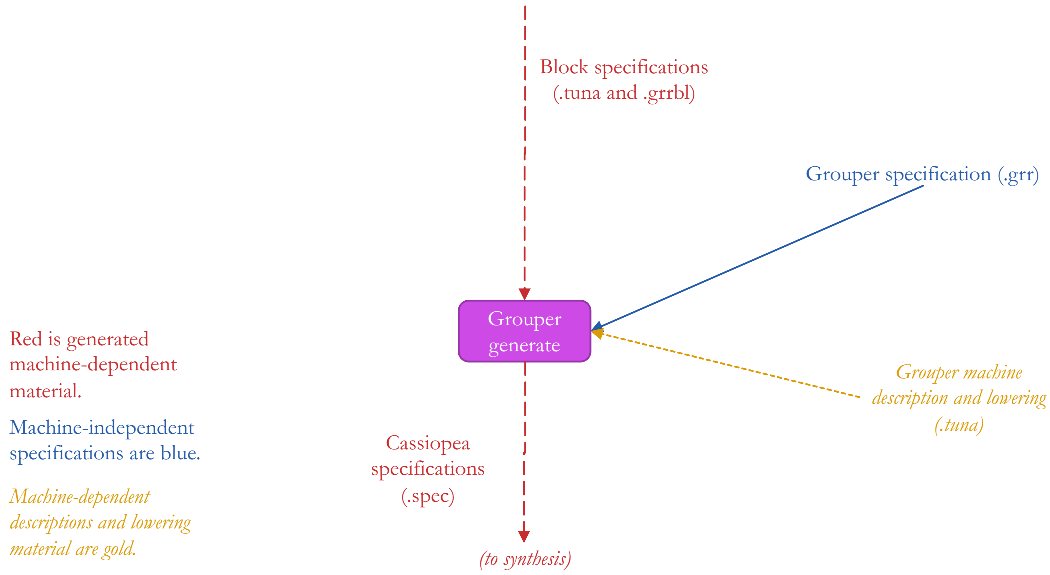
Block specifications (.tuna and .grrbl)
Grouper specification (.grr)
Grouper generate
Red is generated machine-dependent material.
Machine-independent specifications are blue.
Machine-dependent descriptions and lowering material are gold.
Grouper machine description and lowering (.tuna)
Cassiopea specifications (.spec)
(to synthesis)

Figure 5.5: *Grouper generate flow diagram; expansion of the "Producing problem specifications" stage.*

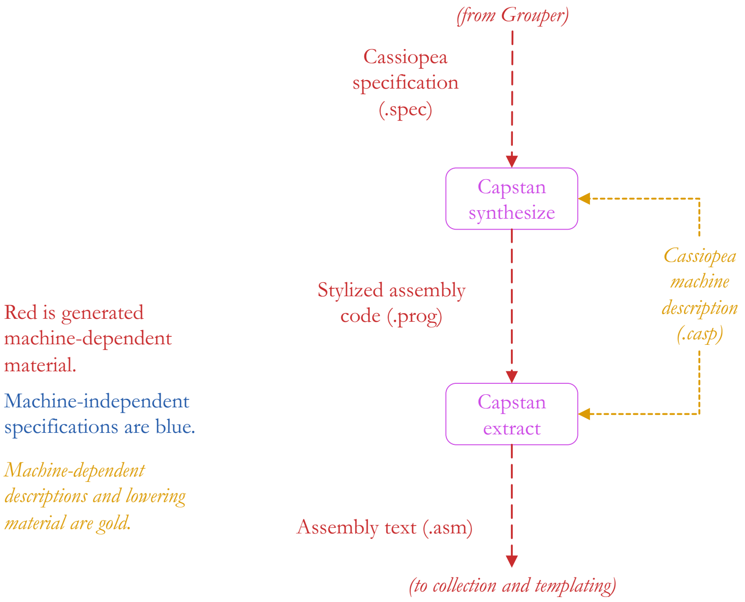
(from Grouper)
Cassiopea specification (.spec)
Capstan synthesize
Cassiopea machine description (.casp)
Stylized assembly code (.prog)
Red is generated machine-dependent material.
Machine-independent specifications are blue.
Machine-dependent descriptions and lowering material are gold.
Capstan extract
Assembly text (.asm)
(to collection and templating)

Figure 5.6: *Synthesis flow diagram; expansion of the "synthesis" stage. Capstan mostly works with a stylized representation of assembly code to avoid needing to know how to parse different assembly languages.*

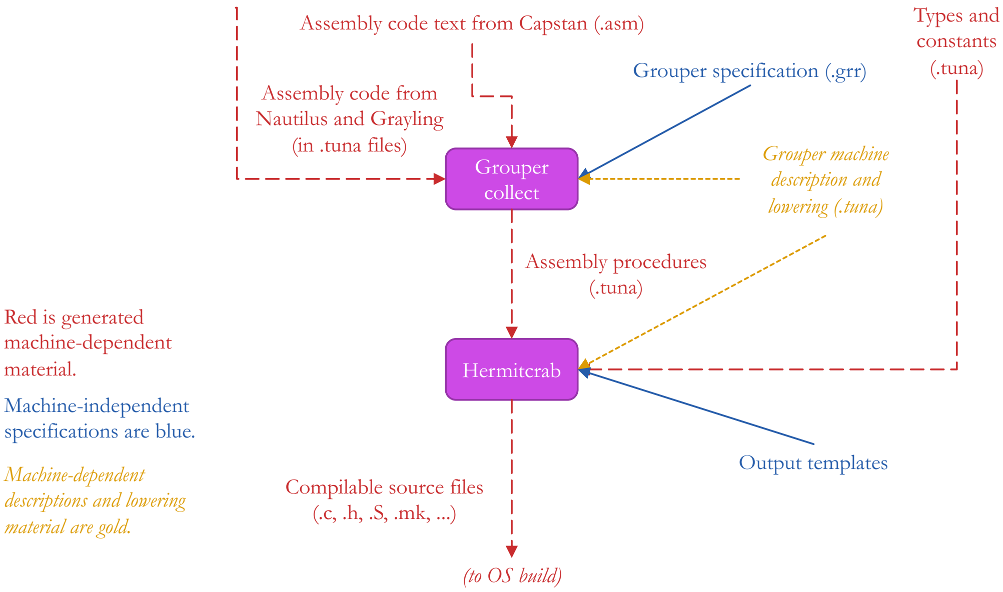
Assembly code text from Capstan (.asm)
Types and constants (.tuna)
Grouper specification (.grr)
Assembly code from Nautilus and Grayling (in .tuna files)
Grouper collect
Grouper machine description and lowering (.tuna)
Assembly procedures (.tuna)
Red is generated machine-dependent material.
Hermitcrab
Machine-independent specifications are blue.
Output templates
Machine-dependent descriptions and lowering material are gold.
Compilable source files (.c, .h, .S, .mk, ...)
(to OS build)

Figure 5.7: *Grouper collect and Hermitcrab flow diagram; expansion of the "Collection and templating" stage.*

cations for Grouper, and types and constants, Grayling can also provide Cassiopea versions of its output code and specifications that can be fed to the synthesis engine Capstan for verification. (The Capstan box is not filled because Capstan is not a topic of this dissertation.)

Figure 5.5 expands the Grouper generate stage, in which synthesis specifications are produced. Figure 5.6 expands the synthesis stage of the top-level figure (Figure 5.1) showing how code is produced, and finally Figure 5.7 shows how the output blocks are combined first into procedures, then files, and then installed in the OS.

The remainder of this chapter briefly discusses each of the elements that does not warrant its own chapter.

5.5 Tuna Files

The Tuna language is a common data interchange format for types and constants. The full specification may be found among the artefacts, in the Grouper tree's documents. The following summary should be sufficient to follow the descriptions in this dissertation. An example appears in Figure 5.8.

Types

The available types are booleans, strings, multi-line strings, bitvector constants (machine integers) of width k, registers of width k, sets of registers of width k, monomorphic sums of products with named constructors, structures comparable to C `struct`s, tuples of arity at least 2, a polymorphic option type like OCaml's, and functions. There are also type aliases (variables that range over types). A unit type (akin to `void` in C) is provided to allow encoding the types of functions that take no arguments or return no value. Tuples and `option` are special-cased to be polymorphic; everything else is monomorphic.

Multi-line strings are distinguished from single-line strings to help improve the formatting of output code.

Values

Values comprise boolean and integer constants, string and multi-line string constants and format strings, names of registers, pointer literals, register set literals, tuples of values, data constructors

```
# Some basic definitions
var machine: string = "mips"
var elf_machine: string = "MIPS"
var need_64bit_millicode: bool = true

# Assembly template for marking a symbol global
var asm_globl: string -> multiline = <"
   .globl $1
">

# Canned unconditional jump instruction; the target label is an argument
var insn_jump: string -> multiline = <"
   j $1
">

# VM page size (base size, ignoring any superpages)
var pagesize: 32 bit = 4096: 32 bit
# Get the copyin_forms type
import specdir vmbase.tuna
# How to do copyin/copyout (picks one of the cases)
var copyin_form: copyin_forms = DIRECTACCESS

# Register initialization for userlevel startup code
import machinedir regs.tuna
region gpmem: 32 bit [1] symbol _gp
var gpaddr: 32 bit = [gpmem, 0x00000000]
var crt0_init_values: 32 register -> 32 bit = [
   gp -> gpaddr,
   ra -> 0x00000000
]
```

Figure 5.8: *Example Tuna file. This is a collection of elements from several of the Tuna files in the Goldfish MIPS machine description and lowering files, including some of the most basic definitions, an assembly language template, a canned instruction, some virtual memory declarations, and a register map giving machine-dependent initializations for the userlevel startup code. The specifications these parameterize are discussed in Chapter 8.*

(which may be partially applied), and finite maps. (Format strings, finite map values, and partially applied constructors have function type.) Pointer literals are a pair (memory region name, offset) and have the same semantics as in Grouper and Cassiopea. See Section 7.2.3 for further discussion.

A *format string* is a string literal that contains at least one `$`. Each substring `$i` (numbered starting at 1) is to be replaced with the *i*th argument. Format strings serve as functions. When fully applied a single-line format becomes an ordinary string, and a multi-line format becomes a multi-line string. The substitution is akin to `printf` in C, but uses numbered arguments so that applying a machine-independent argument list in a machine-independent file (such as a Grouper or Hermitcrab specification) does not constrain the way a machine-dependent format string can be written. The `insn_jump` variable in Figure 5.8 is a multi-line format string.

Finite maps in principle may have any argument type but, in practice, are restricted to register names and strings. The `crt0_init_values` variable in Figure 5.8 is a register map.

Register names are different from variables of register type, like in Grouper and unlike in Cassiopea.

Declarations

Declarations may declare sum types, structure types, type aliases, registers, memory regions, and variables, which are given values. Because register names in the language must be identifiers, and register names in assembly languages are often marked with punctuation (e.g. `%eax` on x86), an additional assembly-language name can be given. Memory regions have the same semantics as in Grouper and Cassiopea. See Section 7.2.3 for further discussion. Declarations may also import from other Tuna files. A Tuna file is a sequence of declarations.

5.6 Nautilus

Nautilus is a tool for producing function call blocks and prologue/epilogue blocks for procedures with C linkage. It is a hack that extracts these blocks by abusing the inline assembler functionality of `gcc` and `clang` to delimit the interesting code and discover the registers holding values of interest.

Nautilus block specifications are function calls and declarations written in C. The full C declaration and expression syntax is supported; this allows calling any C function (and using any type) declared in any header file belonging to any piece of the target system, and allows perhaps-symbolic constant arguments to be placed in the Nautilus specification rather than in assembly code blocks.

Nautilus outputs a Tuna file containing the requested code blocks and the types of the functions involved. It also outputs a `.grrbl` (Grouper block specification) for each code block. (See Section 7.4.1 for further description of `.grrbl` files.)

I only actually ever wrote enough of Nautilus to be sure it would work. Currently it parses its input, generates a C file to feed to the compiler, and runs the compiler, but does not parse the results or actually create its output files. The Nautilus outputs I am using with Goldfish are either collected by hand from this, or handwritten entirely. This is not an ideal state of affairs, but it is not worth spending significant amounts of time on. It should really be replaced by a respectable implementation with a principled model of calling conventions.

5.7 Hermitcrab

Hermitcrab is a simple string substitution language. It imports from Tuna files, converting everything to strings and multi-line strings, and uses the same format string substitution mechanism. It supports refinement (by matching on sum types imported from Tuna files) and some amount of predicate abstraction (by importing variables by name from tuna files) so the input specifications can be machine-dependent. It is not a research artefact, but nonetheless powerful and useful.

5.8 Future Work

Several additional tools were planned (and given fish names) but have not been implemented. I will not discuss them further except for one: an optimizing postprocessor for Capstan and Grouper output.

We would like to have a post-processor to apply simple peephole optimizations to the synthesized code. This could be built with standard techniques and would be a straightforward implementation exercise. (It could even be mostly taken from an existing compiler backend, though most compilers do not expose an interface at the level needed and would also need to learn about control registers and control instructions.) We have observed that this would be useful, but have not actually built it. It would have several purposes.

First, while the synthesis methodology used by Capstan produces the shortest code block satisfying the specification, and that is likely to also be the fastest or best available, Capstan intentionally works from simple machine models to minimize its search space. A simple optimizer can afford to know about many more instructions and instruction variants (particularly on CISC architectures with many addressing modes) and could potentially improve upon Capstan's instruction selections.

Second, procedures generated by Grouper are made up of independently generated blocks. Optimization opportunities will sometimes arise at the boundaries between blocks. Grouper makes no effort to handle these, on the grounds that they would be best handled by this tool.

Third, and similarly, Grayling specifically does not attempt to generate instructions that store multiple registers at once (as available on e.g. ARM) – instead it issues instructions that are meant to be combined by this tool downstream.

Unfortunately, this tool never got a fish name and so I shall refer to it subsequently as just "the optimizer".

(We have also occasionally discussed, but never implemented, a superoptimizer mode for Capstan; that would be a different component.)

Chapter 6

Compiling Context Operations

As mentioned briefly in Chapter 4, saving and restoring registers is a particularly difficult problem for program synthesis. There are two reasons: first, the code blocks for context operations tend to be long; and second, context operations are particularly prone to symmetries that make synthesis harder.

The kernel entry code for an interrupt or exception will, in general, need to save all the general-purpose registers and usually also some control registers. On a machine with 32 registers, that means roughly 30-40 instructions, and then as many again (roughly speaking) to reload before returning to the interrupted context. (While some machines, notably ARM, have multiple-register load and store operations, most do not.) Even if the save and restore blocks are handled separately (as discussed in Chapter 7), this is still an order of magnitude larger than the current scaling limits.

Meanwhile, synthesis becomes increasingly resource intensive when the synthesis problem contains symmetries. ("Symmetries" here means ways in which rearranging elements of a potential solution yields another equivalently correct solution.) This makes the search space much larger than needed; the solver behind the synthesis engine then wastes time and memory considering each permutation separately. Saving a collection of registers (or loading them back) is inherently highly symmetrical: the registers can be handled in any order. With 32 equivalent general-purpose registers any of the 32! possible orderings of the save and load instructions is a valid implementation[1]. Control registers add to this problem: they typically require a general-purpose register as an

[1] It is perhaps ironic that the architecture trends of recent decades intended to make compilation easier also serve to make synthesis harder.

intermediate step, and any general-purpose register whose value has been saved will serve equally well. Furthermore, while in some cases synthesis performance can be increased substantially by hiding most of the registers from the synthesis engine [16], this is not a viable tactic for blocks that save or load context because these inherently involve many registers. (In the case of interrupts and exceptions, they involve all or nearly all registers.)

Meanwhile, these operations are relatively common in kernels. My test platform Goldfish (see Chapter 8) contains six: three pairs of save and restore operations. These are: exception/interrupt handling, thread switch within the kernel, and the C standard functions `setjmp` and `longjmp`. A production OS will have more; for example there will be context switch code for floating point and/or vector units, and potentially also user-level thread switching and signal handler return. These all involve handling different sets of registers.

There is a further related issue: types. For most of these operations, the saved register context must be available as a C data type. This type will have one field for each register involved, so the definition is machine-dependent. These definitions must be generated somehow as part of porting an OS. Doing so is obviously closely related to generating the code, but it is a different problem from code synthesis. Our code synthesis engine was never intended to be able to do it.

For these reasons it seemed like these particular blocks were a good target for a special-purpose tool. I have developed such a tool; it is called Grayling. It has two functions: (1) it generates assembly-language code blocks for context saves and restores, and (2) it also generates C data structures corresponding to the memory access footprint of these operations.

To be clear, it does not generate *whole* context switch sequences: it generates single code blocks that save or restore groups of registers. A complete context switch operation requires composing these blocks with others that are easier to synthesize. (See Chapter 7 regarding composition, and Section 8.3.7 for a discussion of how the complete kernel thread switch is constructed from Grayling-built context blocks and additional synthesized blocks.)

6.1 Grayling

Grayling is a compiler, not a synthesis engine. Its block specifications, though simple, are executable, so the distinction proposed in Section 3.3 makes it a compiler. Grayling takes three inputs:

- A machine-independent problem specification. This can define types, constants, and code blocks. Figure 6.1 gives an example.

- A machine description. This specifies registers, register sets, the sizes and alignments of C data types, and move instructions. Figure 6.3 gives a partial example.

- A "locations" file. This tells Grayling (using machine-dependent register names) where the inputs and outputs to the block are or should be (respectively) placed. These are small; an example appears in Section 6.4.

When run, Grayling reads its problem specification file and outputs a definition from it, which can be a constant, a type, or a code block. For types, and constants derived from them, this involves populating the context structures defined and then extracting to a Tuna file. (Hermitcrab, the templating tool, can then insert these elements into C header files.) For a code block, Grayling compiles the block and produces four different outputs:

- The output assembly code, as a multi-line string constant wrapped in a Tuna file. This is expressed in the target machine's normal assembly language syntax and used by Grouper when integrating context blocks into larger procedures. This Tuna file also declares the types of the block's input and output values for use by Grouper.

- The output assembly code, as a Cassiopea-style program file. This is in S-expression syntax so it can be read by the synthesis engine Capstan for verification. See Section 6.9.

- A formal specification for the output, as a `.grrbl` file to be read and integrated by Grouper. See Chapter 7 for further information.

- (Mostly) the same formal specification for the output, as a Cassiopea `.spec` file to be read by Capstan for verification. See Section 6.9.

The contributions of Grayling are: (a) how simple it can be and still be effective, thanks to compositional code generation; (b) that this simplicity enables compiling something traditionally uncompilable (context switch code has always been written in assembler because it cannot be written in higher-level languages); and perhaps (c) the combination of techniques that allows Grayling to avoid needing additional scratch registers.

```
context switchframe {
   calleesave;                  # callee-save registers
   returnaddress;               # any return address register(s)
   exclude currentregs;         # but not any curthread/curcpu registers
}

block storeregs {
   arg ptr: struct switchframe * in storemem;
   store(ptr);
}

block loadregs {
   arg ptr: struct switchframe * in loadmem;
   load(ptr);
}
```

Figure 6.1: *Grayling specification for thread switch.*

6.2 Specification Language

An example problem specification, for thread switch, is shown in Figure 6.1. It declares a context structure (`switchframe`), which contains all callee-save registers and the return address register, if the machine has one, but not the register that holds the current thread or CPU pointer, if any, which needs to be handled differently. It also defines two code blocks, one to store the registers and one to load them. Each takes a `switchframe` pointer as an argument (input) and loads or stores the registers through that pointer. In the store block, the pointer points to a memory region called `storemem`; in the load block, the memory region is called `loadmem`.

The abstract syntax for Grayling specifications is shown in Figure 6.2. Let k be integer constants, w be integer constants that are specifically bit widths, and x be identifiers. (Some uninteresting details have been omitted for clarity, e.g. different flavors of `sizeof`. These may be found in the source, in the Grouper tree among the artefacts.) Types t are booleans, bitvectors, pointers, and context structures. The type x `bit` is a bitvector of the same bit width as the type x. (These types can come from the machine description as well as the specification; this permits using machine-independent abstract sizes in problem specifications.)

Expressions e are intentionally quite limited – constants, variables, pointer dereference, equality test, and size extraction. These are used to identify the location to save to (or restore from) and for constructing values for export to other tools.

Statements s are even more limited: you can store context to a pointer expression (which must

Grayling Types

$$t ::= \quad \texttt{bool} \mid w\,\texttt{bit} \mid x\,\texttt{bit} \mid \texttt{pointer}\ t \mid \texttt{context}\ x$$

Grayling Expressions

$$e ::= \quad k :\ w\,\texttt{bit} \mid x \mid \ast e \mid e_1 == e_2 \mid \texttt{sizeof}\ t$$

Grayling Statements

$$s ::= \quad \texttt{load}\ e \mid \texttt{store}\ e$$

Grayling Declarations

$$\begin{array}{ll}
\ell ::= & r_s \mid \texttt{exclude}\ r_s \mid \texttt{extra}\ x : w\,\texttt{bit} \\
\mid & \texttt{zeropadding} \\
d ::= & \texttt{type}\ x\ =\ t \mid \texttt{const}\ x\ =\ e \\
\mid & \texttt{block}\ x_i \ldots\ x_o \ldots\ s \ldots \\
\mid & \texttt{context}\ x\ \texttt{"abc"}\ \{\ell \ldots\}
\end{array}$$

Figure 6.2: *Grayling abstract syntax. w is a bit width (integer literal), x is an identifier, r_s the name of a register set, and x_i and x_o are names for block inputs and outputs respectively. ℓ is an element of the context declaration.*

point to a context type) or load it. This transfers all of the registers (and any extra elements, described below) from the CPU to memory or back. These are all the things that can be done in a block, and this is all that is required. Note that while a single block might load or store more than one context type, because this falls naturally out of the implementation, blocks that do both loads and stores at once introduce complications and are not fully supported. (Neither of these cases arises in normal operating system code.)

Declarations d allow declaring types and constants for export. (Declaring a type alias this way allows exporting a machine-independent name for a machine-dependent type.) The elements of a block declaration, as shown in the figure, are input names x_i, output names x_o, and a list of statements s. Inputs may be values for extra elements to be saved into a context structure (these are matched by name) or used to name the location to load or store to. Outputs may be extra elements to be restored from a context structure (also matched by name) or the names of inputs, if the block should preserve those values. The registers used for inputs and outputs are defined by the locations file.

A context structure declaration always has a name x, and optionally a prefix string `"abc"`. This is inserted in front of the register names in the output type declaration; it produces the common prefix on the member names of structure types that is often seen in traditional C code. It then contains some number of further elements ℓ, which may be:

- A machine-independent register set name to include in the context.

- A machine-independent register set name to exclude from the context.

- Zero or more "extra elements", each a name x and a type (that must be a bitvector type) that are added to the context structure. These must be passed in as inputs (or out as outputs) of blocks manipulating the context structure. (They are matched by name, so need not be explicitly manipulated.)

- A flag `zeropadding` – if applied, Grayling makes sure that stores to the structure zero any padding bytes. This is necessary in some contexts in kernel code to avoid information leaks, but is otherwise a waste of time. (See below for further discussion of padding.)

The set of registers given fields in the context structure, and saved or restored by the output code blocks, is the union of all register sets included, less (by set difference) the union of all register sets excluded. Register sets are defined in the machine description.

A typical specification, such as the example in Figure 6.1, will have one context type, one block each for saving and restoring it, and perhaps some additional constants such as the size of the context type.

Notice that this language is all highly specialized to Grayling's task, and also to some extent to Grayling's environment. Grayling assumes it does not have to generate whole procedures, and that other code blocks can and will be placed around the code it generates. This avoids the need to consider a wide range of circumstances that might otherwise need special handling. For example, if one needs to save a context to a global variable, the context block would be preceded by a synthesized block that loaded the address of that variable into a register. This means that Grayling needs no handling whatsoever for assembler/linker symbols.

The ability to insert extra elements into contexts is needed to support the Standard C functions `setjmp` and `longjmp`, where the return address from `setjmp` must be stored in the context for later use by `longjmp`. On some machines the return address appears in a register and could be captured directly by Grayling; but on most it needs to be read off the stack. Rather than insert special-case logic, I provided a mechanism that allows the specification to use the existing mechanism for C linkages (Nautilus) to collect the return address beforehand and then pass it into the Grayling block, and then in `longjmp` extract it from the Grayling block and let Nautilus put it in place for the return.

6.3 Machine Description

Fragments from the 32-bit MIPS description are shown in Figure 6.3. (The complete description may be found among the artefacts, in the MIPS descriptions in the Goldfish tree.) The byte size is 8 (of course), the maximum alignment is 4, pointers are 32 bits wide and so are the C types `unsigned` and `size_t`. Allowing for byte sizes other than 8 is probably frivolous, but also harmless. The alignment requirements, however, are needed for generating structure padding. (Padding is discussed in more detail below.) The alignment requirement for each type defaults to the size if not given. Note that Grayling does not (currently) support exotic machines where pointers to `char` are wider than other pointers, although adding such support would not be particularly difficult.

Three general registers and one control register are shown. `z0` is the always-zero register, which is used to streamline zeroing; `t0` and `t1` are ordinary general-purpose registers. `c0_cause` is a control register, the one that reports what happened when a trap occurs. Each of the general registers has an assembler name (this is the string used to output assembly language text) and a Cassiopea name (the name used to output text to be read by Capstan for verification). The latter can be different for the specifications and the code; this is needed here due to the encoding of the control register access instructions in the Cassiopea machine description. The property `caspspecusefunc` tells Grayling to read the register by calling a Cassiopea-level function instead of using the ordinary read-register syntax. This is needed for control registers with subfields, as the preferred way to model them in Cassiopea is to make each subfield a separate register and use functions and procedures for access to the whole. The `readonly` property marks registers that should be saved but not restored.

Registers also have the following properties not shown in the example:

- An index number.
- A validity predicate for values.

The index number is used to define the “natural” or “normal” ordering of registers in the machine, which is used to sort the fields of context structures. By default the index number is taken from the ordering of register declarations in the file.

The validity predicate is needed for registers where the possible values are restricted. For example, it is not uncommon for control registers to have always-zero bits, where writing ones to them is ignored. The validity predicate is made a precondition of load blocks, so that only valid values are loaded into the register. (Otherwise, the value found in the register after the load might

```
charbit 8;
maxalign 4 bytes;
ctype pointer: 32 bits;
ctype unsigned: 32 bits;
ctype size_t: 32 bits;

reg z0: 32 bit {
   asmname = "$0";
   caspname = "r0";
}
zeroreg z0;
reg t0: 32 bit {
   asmname = "$8";
   caspname = "r8";
}
reg t1: 32 bit {
   asmname = "$9";
   caspname = "r9";
}
reg c0_cause: 32 bit {
   asmname = "$13";
   caspname = "0b01101";
   caspspecname = "cp0_13_cause";
   caspspecusefunc = true;
   readonly = true;
}

regset generalregs { t0, t1, ... }
regset allgeneralregs = { z0 } + generalregs;
regset cop0regs { c0_cause, ... }

insn {
   asm "lw ${rd}, ${off}(${rs})";
   prog "(LW ${rd} ${rs} ${off})";
   from (rs + off) where rs in allgeneralregs,
      -32768 <= off: 16 bits <= 32767;
   to rd where rd in allgeneralregs;
}
insn {
   asm "mfc0 ${rd}, ${rs}";
   prog "(MFC0 ${rd} ${rs} 0b000)";
   to rd where rd in allgeneralregs;
   from rs where rs in cop0regs;
}
```

Figure 6.3: *Partial Grayling machine description.*

not match the value placed in it, which causes verification to fail.) The validity predicate is also made a precondition of store blocks, so that values read from the register can be loaded back later in a hypothetical future environment where the kernel is verified and this property can be carried through from one block to the other. As a matter of expedience the validity predicate is handled as a pair of strings, one in Cassiopea syntax for verification and one in Grouper syntax for Grouper integration. It is left to the user to keep these mutually consistent. (This is not difficult in practice.)

The example also shows three register sets and two instruction templates.

The first instruction template describes the `lw` (load word) instruction; it gives the output syntax for both assembly and Cassiopea (`prog`) forms. These define three operands, `rd`, `rs`, and `off`. The `from` clause defines where the instruction reads from (in this case, the address found in `rs` plus the given offset) and gives constraints on the allowable values for the operand. In the case of `rs`, it must be in the `allgeneralregs` register set. The offset, which is 16 bits wide, must be within the range of a 16-bit signed value. The `to` clause similarly defines where the instruction writes to. All instructions are moves; that is, they copy values from one location to another.

The second instruction template defines the `mfc0` (move from coprocessor 0) instruction that is used to read control registers.

The general concrete syntax for from and to locations is an expression (using addition, multiplication, parentheses for indirection as often used in assembly languages, operand names, register names in quotes, and constants) followed by constraints that give types (register or integer) and register set inclusion or bounds predicates.

The machine description is a somewhat ad hoc collection of things Grayling needs to know. Many of these could be provided a different way (such as via the same Tuna files that Grouper uses to configure itself) – there is a lot of room for future work on implementation tidiness.

It is better thought of as a collection of attributes and tables than an abstract syntax; it has the following elements.

- The bit size of bytes (normally 8).
- The maximum alignment requirement.
- The byte size and alignment of pointers.
- C type names with byte size and alignment.
- Registers and register sets.
- The name of an always-zero register, if any.

$$
\begin{array}{lr}
p_r ::= & \texttt{fixedreg}\ r \\
| & \texttt{regset}\ r_s\ x \\
| & \texttt{anyreg}\ x \\
 & \\
p_i ::= & \texttt{fixedint}\ k \\
| & \texttt{boundedint}\ k_1 \leq x \leq k_2 \\
| & \texttt{anyint}\ x \\
 & \\
m ::= & \texttt{justreg}\ p_r \\
| & \texttt{regoffset}\ p_r\ p_i \\
| & \texttt{regscaleoffset}\ p_r\ k\ p_i \\
| & \texttt{stackoffset}\ p_i \\
 & \\
p ::= & \texttt{reg}\ p_r \\
| & \texttt{mem}\ m
\end{array}
$$

Figure 6.4: *Grayling instruction parameterization representation. p_r is a register parameter; p_i is an integer parameter; m is an addressing mode (memory reference); p is any parameter. x is the name of an operand in the instruction syntax, r is a register name, and r_s is a register set name. All instructions are moves; the top level parameters for each instruction are `to` and `from`.*

- Instruction templates.

This is converted internally to the representation shown in Figure 6.4, in which p represents a parameter for the use of the instruction, either the from-location or to-location. This representation is more suitable than the concrete syntax for matching against moves to be performed.

For example, if Grayling wants to move a value to the register `t0`, it searches the instruction templates for one where the destination has the form `reg` p_r, and p_r is one of

- `fixedreg` `t0`
- `regset` r_s x where `t0` $\in r_s$
- `anyreg` x

If the form it finds contains an operand name x, that operand name is bound to `t0` and the proper printing form of `t0` is substituted into the instruction syntax. If there is no x, the instruction implicitly uses that register and no operand needs to be generated. The matching process for memory locations is similar. The supported addressing modes are:

- `justreg` p_r - read from the address in a register
- `regoffset` p_r p_i - read from a constant offset to a register
- `regscaleoffset` p_r k p_i - similar but multiply the offset by k first
- `stackoffset` p_i - like regoffset but specifically relative to the stack pointer

The stack pointer is special-cased as part of (currently incomplete) direct support for push and pop instructions that update the stack pointer as well as moving values. Adding support for handling addressing modes (such as using one register as an offset for an address in another) is not difficult. The logic to take advantage of these addressing modes so they can be found and used for move operations is less so. (For example, the logic for carrying the offset along in an additional register, so as to use the two-register addressing mode, is not entirely trivial, not implemented, and left for future work.)

Note that because registers may be general-purpose registers, special registers, control registers, or others (e.g. floating-point or vector registers) the `anyreg` form is mostly not that useful.

Instruction templates are not limited to single instructions; in some cases it might be useful to use sequences of multiple instructions. (In fact, the original design was to attach a template for the full load and store sequence to each register, including any intermediate steps. These templates would be synthesized, instead of needing to be specified directly. This plan was dropped for two reasons:

first, it turned out to be readily possible to have Grayling figure out multi-stage moves on its own, without needing to resort to solving or synthesis; and second, the synthesis engine cannot currently handle specifications that are parameterized or include symbolic constants. This would mean that a new set of templates would need to be synthesized for every different structure offset registers were written to, and this rapidly became unappealing.)

6.4 Locations

The locations file gives locations for each block's inputs and outputs. To that end, it is a list of identifiers (the input or output name) paired with locations, where a location can be either a register name, an offset from the stack pointer, or an offset from some other pointer. In the latter case the other pointer must be a named input and a location for it must also be provided. (The stack pointer is distinguished because instructions that use it often change it, in which case offsets need to be adjusted to compensate. There is currently only partial support for such instructions.) Locations are required to be disjoint, so this is a very simple form of separation logic.

An example locations file suitable for the example specification from Section 6.2:

```
ptr: register sp
```

This means that the value for the `ptr` input will be found in the `sp` register.

6.5 Producing Context Structures

There are three primary considerations when generating the definition of a context structure: membership, ordering of the elements, and structure padding. After attending to these concerns, Grayling outputs the type definition to a Tuna file for subsequent handling by other tools, mostly the templating tool Hermitcrab.

6.5.1 Membership

A context structure inherently has one member for each register it is supposed to save, whose type is the same as the type of the register. The registers are specified as a list of register sets to include and a list of register sets to exclude. (There has been no need to support other set operators.) Additional elements may be introduced, as discussed above. The machine description contains declarations of C types and their sizes. Grayling uses a reverse index on this list to pick a C type for each structure element. (If there are multiple types of the same size, such as `int` and `long` on ILP32 machines, it picks one arbitrarily. Note that the purpose of declaring `int` and `long` in the machine description is not to declare that these types exist, because they always do, but to declare what size they are, as this can vary.)

6.5.2 Ordering

While for the most part the order of the elements in a context structure is completely irrelevant, there are four reasons to allow controlling the order:

- On machines with multi-register load and store operations, the order of the registers in memory is typically fixed. While Grayling does not issue these instructions itself, it is intended to issue sequences of single loads and stores that can be combined into multi-register operations by a downstream optimizing pass. Therefore, it should make an effort to lay out context structures in a way compatible with such instructions.

- It is also often desirable to access the context structure in memory order; in particular when there are special-purpose push and pop instructions for stacks, they are often faster. Grayling also cannot currently issue these instructions itself, but converting a stack allocation followed by sequential accesses to offsets to a series of push or pop instructions is a reasonably straightforward downstream optimization.

- In light of this, since loading or storing a control register typically requires using a general-purpose register as a temporary, we also want to be sure to sort the structure so that general-purpose registers are stored first and loaded last. Then the code to handle any control registers can avoid needing extra general-purpose registers to use for scratch space.

- On some machines for some context operations, part of the machine state is saved by the hardware before the Grayling code runs. (For example, on kernel entry on x86, the hardware pushes the user-level stack pointer onto the kernel stack, and loads the kernel stack into the stack pointer register.) Ideally these elements should be placed such that if the Grayling code did save them it would save them first, and in the same order; then the context structure can be placed to overlap the already-saved entries and the code for those saves can be omitted. Unfortunately, this is not implemented yet.

On normal machines where stacks grow down, the “far end” of the context structure (largest offsets) is saved first and loaded last. Therefore the sort order is as follows:

1. First, any extra entries.
2. Second, control registers (and special registers).
3. Third, general purpose registers.
4. Fourth, any presaved registers.

with ties resolved by the machine register order declared in the machine description. It has not so far been necessary to allow the order of presaved registers to vary across context types belonging to the same machine. One would not expect it to, as typically these exist only for traps (interrupts, exceptions, system call entry points, etc.) and for hardware reasons the order is likely to be consistent across the various kinds of trap. Grayling does not currently support machines where stacks grow up (such as the HP PA-RISC) but adding this support would be a fairly straightforward piece of future coding work.

6.5.3 Padding

C structure types can contain *padding*, extra/waste space that the compiler inserts so that the elements are properly aligned according to the requirements of the machine. (And sometimes also according to efficiency dictates.) As mentioned above, in some situations[2] it is important to prevent leftover garbage from previous uses of the same memory from leaking out. It is also critical for Grayling and the C compiler to agree on the byte offset of each structure member.

Unfortunately, C compilers are free to do as they please with structure padding, rather than necessarily inserting only minimal padding. Fortunately real compilers (as opposed to hypothetical compilers proposed by `comp.lang.c` aficionados like myself) do not in practice abuse this latitude, and insert minimal padding, so that Grayling can replicate the compiler’s behavior with a high degree of confidence. Also fortunately, the results are easily cross-checkable. See Section 6.9 below for the methodology.

Grayling inserts padding when computing the byte offset for each structure element: if it encounters an element that is inadequately aligned, according to the required alignment specified for

[2] One might say “in some contexts...”

the type in the machine description, it inserts explicit padding fields. Grayling always makes padding explicit, because this allows both it and the alignment requirements in the machine description to be conservative. C compilers are not allowed to remove structure members, so will not remove unneeded explicit padding; however, overestimating the size of an implicit padding field would cause the computed offsets to be incorrect.

6.6 Code Generation

There are a handful of considerations that drive the architecture of the code generator.

First, on most machines at least some registers (typically control registers or special registers) cannot be stored directly to memory; they need to be moved into a general-purpose register first. This requires that a general-purpose register be available; that is, in general, the value in that register must have been saved already, or not loaded yet. (Some context operations, such as thread switch, need to save only some registers, so that availability of additional general-purpose registers to use as scratch space is not problematic. However, the store and load operations for trap handling are more constrained: they must in general save and load (respectively) every register, so scratch space must be considered carefully. Note that I assume every register that needs to be handled by a context operation is only one move step away from a general-purpose register. Handling longer chains of moves is possible, but requires more analysis of the specification, and is not worthwhile since such move chains do not occur in real machine architectures.

Second, while in most save operations the registers saved are about to be overwritten and the values in them can be destroyed, this is not always the case. The C standard function `setjmp` does a context save but must nonetheless preserve, as well as save, the callee-save registers. This led me to provide a switch to turn off the scratch register handling entirely. (Doing so is sufficient for `setjmp` as long as no machine arises where control registers or special registers with no direct route to memory are among the callee-save registers. I know of no machine where this is the case.).

Third, reading and writing control registers is not always effect-free; each control register that is to be accessed should be accessed only once. For ordinary code blocks that transfer one set of registers to memory, this is not an issue. However, because overengineering led to excessive generality in the specifications (in particular, context operations that access multiple context structures in one code block) the internal representations include cases where a single register needs to be saved to multiple locations. Handling these cases correctly proved simpler than excluding them.

6.7 Implementation

After reading all the inputs and typechecking them, Grayling extracts an element from the specification. If this is a type alias or constant, it just prints the selected item into a Tuna file. If it is a context structure, it must create the context structure; if it is a code block, it must first create all context structures required and then generate a code block and its specification. Generation of context structures is fully deterministic so that repeated runs will always produce the same type. (Otherwise, the generation of types and code blocks in separate invocations would be unsafe.)

6.7.1 Generating Code

Although Grayling is a compiler, not a synthesizer, its executable specifications are not very specific. It would be reasonable to begin code generation with some form of synthesis, e.g. solving for a series of move operations. (And while ordinary synthesis with a full machine description is not sufficiently scalable, synthesizing using only move operations might be.) However, I chose to avoid such techniques as being unnecessarily complicated. Instead, Grayling generates a list of moves from tables of initial and final locations of values; it produces the tables by a simple form of symbolic execution.

Location tables

The location tables map value identities to lists of locations. They are reversed-indexed so they can be conveniently accessed by either value identity or location. Value identities are either zero, the prestates of specific machine locations (registers or memory), or explicitly named values, which are the input and output arguments of the block. Zero is specifically used when zeroing padding fields. Extending this support for zero to other constants might be useful in the future; e.g. to allow constants as parameters for blocks without needing to place them in a register.

For each value in the initial machine state, Grayling initializes a table entry with the value's identity and its starting location. Then it steps through the statements in the block specification updating a copy of this table by moving (moving in an assembly-language sense, that is, copying) values to additional locations. Loads into read-only registers are ignored. This yields one table for the initial state and a second table for the final state. It then updates the final state table by adding the desired locations of the block's output arguments, and removing the entry for any value whose location is incidental. An incidental location is one that is not the result of an explicit load or store, or an output argument. This amounts to initial locations that have not been overwritten; dropping these allows using registers whose values have been saved as scratch space for moving control registers. (This concept was also originally intended to be extendable to intermediate locations for blocks combining both loads and stores; but this turned out to be both complicated and not useful in practice.)

Move lists

After generating the location tables there are four steps to produce a list of individual move operations:

1. Digest the location tables into a table mapping initial locations to lists of final locations.
2. Sort this table so as to produce operations in the desired order.
3. Convert the lists of final locations to sequences of single moves.
4. Instantiate registers for temporaries.

Digesting the location tables is straightforward.

Sorting before constructing single moves may seem counterintuitive, because it makes the sort criteria complicated and unnecessarily groups all moves of the same value together; but it has an important purpose. It allows using a minimal number of scratch registers by default without having to do complex scheduling or planning.

Recall that the goal is to access the structure fields in the same order that stack push and pop operations would, to enable the use of those operations. Therefore the primary sort criterion is the context structure offset. The sort works by looking for memory offsets in the source and destinations and comparing those if possible, and otherwise dropping back to arbitrary criteria. (This might perform poorly for large destination lists, but this is not an issue in practice.)

Now, converting a source and list of destinations to a list of single moves is simple if each move can be made directly (which can be determined by checking the available code snippets): just move from the source to each destination. If there is no such instruction, however, it gets more complicated. The method Grayling uses is to look for a general-purpose register in the destination list and use that for staging: move the value to there first and then to all the other destinations. If there is none, it allocates a scratch register to serve the same purpose, and drops it when done. At this stage Grayling uses a simple intermediate representation to allocate numbered scratch registers and instantiate them later. This is intended to allow further reordering of the operations by a subsequent pass in cases where sufficient scratch registers are available. (This pass, which would improve the chances of being able to use multi-register operations for complex blocks, is future work, and not necessarily in fact worthwhile as complex blocks do not really occur in practice.)

Note that this logic assumes that moving to a general-purpose register is sufficient to then be able to move anywhere with one more step. This is true in practice for most or all real machines: while special instructions are normally needed to access special registers and control registers, these instructions use general-purpose registers. And while many machines do have state that is two steps away from a general-purpose register, such as cache tags or TLB entries, these are not, in general, part of context operations. It would have been possible to have Grayling do a more sophisticated analysis to allow it to handle arbitrarily long chains of moves (for example, one might partition the machine's locations by how they are accessed, build a graph with edges for all the possible accesses, and look for a minimum spanning tree connecting the source and all the destinations) but this serves no real purpose. Avoiding the pursuit of spurious generality is critical not just in general but specifically for tools and materials related to portability. (Grayling is already overengineered enough in several areas and does not need more of the same.)

The next step is to instantiate the scratch registers, which is done by iterating through the intermediate move list, carrying along a map holding the instatiations and a fresh location table to keep track of the current machine state. We can safely clobber a register if it is a general-purpose register (other registers, in addition to not necessarily being useful temporary locations, can cause unpredictable behavior if arbitrarily scribbled in), it still contains its prestate value, and either (a) its prestate value has been moved to at least one memory location, or (b) its prestate value is ultimately not used. When loading registers, this skips registers that have already been loaded (they no longer contain their prestate value) and allows registers that have not (their prestate values are not used). When saving registers, this skips registers that have not been saved yet (though they contain their prestate value, it has not been moved to memory) and allows registers that have been saved. It does not work for blocks that both load and save registers, which is why such blocks are not supported. A more sophisticated usage analysis would be required to support these. Note that this logic also depends on the batching behavior of the prior steps: because scratch allocations happen only between groups of moves with the same original source, once a value has been written to any memory location, at a scratch allocation point it has already also been written everywhere else it needs to be. A more sophisticated usage analysis could also lift this restriction.

Scratch registers are also explicitly limited to the set of registers involved in the context operation. The ordering of the operations (based on the ordering of the structure elements, so that general-purpose registers are stored first and loaded last) means that as long as the context operation involves at least one general-purpose register, which is normally the case, there is always at least one general-purpose register available as a scratch register, and its use as a scratch register is effectively invisible to surrounding code. This is a highly desirable property, because one of the places where Grayling code is specifically needed is saving registers on entry to a trap handler, where all values from before the trap must be preserved, and there is extremely limited scratch space available. (And that space is likely to be needed for other things.) It also means that there is no need to arrange a mechanism to negotiate scratch registers with surrounding code or with Grouper on behalf of surrounding code.

The register allocator also prefers caller-save registers; this is not for correctness (the above logic is sufficient) but to avoid the situation where (for example) it picks the return address register as a scratch register, which is confusing and unnecessarily alarming to anyone reading the output code.

Instantiating instructions

The final step in code generation is to convert the list of moves to a list of instruction templates. This is done in the most straightforward way possible: by scanning for a template that matches the source and destination. The number of instructions to scan is not enough that this requires complex lookup infrastructure. (Even if every special or control register has its own instruction, which is not usually the case, the number of instructions is still only tens.)

After picking the register, Grayling substitutes the operands (the source and destination locations) into both styles of output and adds the resulting text to the output files. (Recall that Grayling generates its output in two forms: one to be consumed by Capstan for verification and one suitable for the assembler.)

6.7.2 Generating Specifications

The specification generation is explicitly separate from the code generation; each works independently from the basic internal representation of specifications and machine descriptions (and context structures) – in particular the move list logic is not shared. This is to maximize the probability that a bug in either will lead to inconsistent output that fails verification.

The move list for specifications can be less operational; there is no need, for example, to consolidate moves of the same value. So it can just collect sources and destinations from the load and store operations and the structure definitions they are applied to.

Then for each move it can declare a variable to hold the value and a pair of assertions that place that value in the from and to locations. These are for the precondition and postcondition respectively. There is one complication: in Cassiopea specifications each variable must be initialized to a value, but in Grouper specifications variables are *not* initialized but bound separately by assertions. (This impedance mismatch also affects Grouper's operations, and is discussed further in Section 7.5.2 of Chapter 7.)

The other difference between the two types of specification (besides concrete syntax, which is easily handled) is that the Cassiopea specification is purely for a single block, and the Grouper specification is intended to integrate into a larger procedure. In general, for procedures that involve context operations, the register state is intended to come from (or be delivered to) the start state (or end state) of the procedure, not just to the state of the immediately adjacent program points. For this reason the block specification emitted by Grayling includes assertions for four program points: an initial program point, the program point immediately before the context operation (the same as the Cassiopea precondition), the program point immediately after (the same as the Cassiopea postcondition) and a final program point. When Grouper reads the specification it will rewrite the program point names to match what it has been instructed to do. The initial and final assertions differ from the block precondition and postcondition in that they refer to the state of only the registers and context structures; the precondition and postcondition assertions also refer to the block input and output arguments.

6.8 Partial Saves in Hardware

For some context operations (mostly trap handling) and some machines, some of the values that need to be saved are saved to memory (or to other registers) by the hardware. This happens before software gains control. This is intended to be handled by providing an alternate location for these values and using that alternate location as the source for moves.

The difficult question, from an implementation standpoint, has been where to declare these locations. They are both per-problem (so they do not belong in the machine description) and also per-machine and machine-dependent (so they do not belong in the problem specifications either). Though locations, they do not belong in the locations input file either as that is for placing items whose location is not fixed; the alternate locations for hardware-saved elements are fixed by the machine architecture.

Lack of a ready solution for this problem, menial though it may be, has led to postponing the implementation to future work.

6.9 Verifying the Output

There are two things in Grayling's output to verify. First (and simplest) that the context structure offsets it has computed match the C compiler. Second, that the code it generates matches the specifications it generates. (It would be nice to also verify the specifications... but that involves finding something to verify them against. Fortunately the specification generation code is comparatively simple. In the long term the right approach is to verify Grayling itself.)

6.9.1 Structure Offsets

Verifying the structure offsets (and padding) against the C compiler is relatively simple. One can generate a C file that includes a copy of the structure declaration, and then, using standard techniques for compile-time assertions, checks that the size and offset of each field is what Grayling expected. This file will not compile if Grayling and the compiler disagree on the structure layout. Implementing this feature is, however, left to future work.

6.9.2 Code

As discussed above, Grayling emits two versions of the output code and output specification, one intended for use by Grouper and the other as Cassiopea for verification by Capstan. (They are generated together by printing different output syntax from the same internal representation.) Thus, verifying the output code is as simple as running Capstan in verification mode and passing it the output code and specification, and a Cassiopea machine description for the same machine. The Cassiopea machine description and Grayling machine description must agree for this to work. Minor differences in the representation or naming of instructions and registers in the Cassiopea machine description can be accounted for in the Grayling machine description.

All the Grayling specifications used in Goldfish produce code that verifies on all the architectures with which I have been working, with one exception. The RISC-V trap handling code needs the partial saves feature (Section 6.8) to have a register available to point to the trapframe structure. Without this the store code does not verify.

6.10 Discussion

For dealing with structure padding and structure offsets I chose to have Grayling generate the structure itself and cross-check it against the C compiler afterwards, rather than pass the structure to the compiler and have the compiler hand back the offsets. There were two reasons for this choice.

Although C compilers are *permitted* to do virtually whatever they want with structure padding, in practice, the behavior is simple and predictable. In fact, on platforms where multiple compilers are expected to generate interoperable code (as is the case today with gcc and clang on Unix platforms), these compilers must agree on structure padding, and they do so by implementing the traditional simple and predictable behavior. Thus reimplementing it is low risk.

Conversely, extracting structure offsets from a C compiler is problematic. Essentially all production OSes (and many research OSes) have a tool for this, to allow accessing C struct types from assembly code, but these tools are at best fragile. Often they rely on compiling and running a C program that prints out the offsets. This is problematic in an environment where one is cross-compiling and does not have the OS running on a new target machine yet. Furthermore, running this tool from inside Grayling would cause a number of practical complications for the build process. (Note that Goldfish does not already have such a tool, because OS/161 does not, because it was specifically built to avoid the need for one. If it already had one, this choice might have been different.)

6.11 Future Work

Some context structures are part of the kernel ABI. (For example, this includes the structures for `ptrace` and signal handling, which do not appear in this dissertation but do appear in production OSes.) In most OSes, arbitrary changes to the kernel ABI are not acceptable; any changes to these structures need to be accompanied by versioning, compatibility code, and assorted administrative headaches. The intentional determinism of the Grayling structure generation is only really intended to make structures stable in the short term; for long term stability one would like to have the ability to generate the structure once, check it into version control, and only read it back later, and for intentional changes have a scheme for keeping the old version under an alternate name and generating definitions and code for both versions. Most of the support for such a scheme exists (it just amounts to having more structure definitions and more blocks) but the mechanism for making it work nicely does not.

Chapter 7

Composition of Program Blocks

As discussed in Chapters 4 and 5, the scaling limits of assembly language synthesis dictate that there must be some way to compose synthesized blocks. Recall from Section 4.1 that, in this context, a block is a sequence of only a few instructions. One can think of them as similar to basic blocks, except often smaller: several synthesis blocks might be needed to produce a sequence of straight-line code that would still be a basic block in a conventional compiler.

Experience very quickly showed that writing separate specifications for each block did not scale past a small handful of blocks at a time. When one block follows another, the postcondition of the first block must in general be copied as the precondition of the second. This creates significant consistency problems even when writing the first version of a procedure; the effect on long-term maintenance would be far worse. Furthermore for a large procedure there are multiple and often complex invariants that can be carried across many blocks in succession. Keeping multiple copies of all of these consistent is even harder. Meanwhile, even in OS/161, which is small, there are enough procedures that trying to manage all the individual blocks by hand becomes a challenge. And also, even though many of the low-level procedures in an OS are simple, one often needs control flow. Manually tracking control flow among large numbers of individual blocks piecemeal is effectively impossible. Some tool for managing the blocks and specifications is required.

There are two other motivations as well. First, for various reasons already discussed, we have multiple tools for producing blocks. Each of these has its own quite different concept of block specification. Safe composition of such heterogeneous specifications is not automatic. Second, because we need machine-independent procedure specifications, these cannot explicitly allocate and manage registers, or even know how many registers are available. Some compromise between entirely automatic register allocation (like a compiler) and entirely explicit register managment (like handwritten assembler) is required.

For these reasons I developed a technique, compositional code generation, and a tool (and associated specification language) called Grouper to implement it. Grouper's role is to produce whole assembly language procedures using our various block generation tools, either addressing these challenges directly or providing infrastructure that allows working around them in a reasonably tractable way.

7.1 Grouper Overview

A Grouper specification defines one or more procedures to be generated in assembly language, by a combination of compilation and synthesis techniques implemented in multiple external tools. The specification as written by the OS architect is machine-independent. External machine-dependent definitions (one set per supported machine) serve to specialize the specification to a particular machine architecture.

The Grouper specification defines procedures in terms of blocks separated by program points. Each block is assigned to a particular block generation tool: Nautilus for procedure calls, Grayling for context operations, Capstan for synthesis. Blocks may also be literal – this generally means literal strings imported from the machine description material, not string literals in the specification itself. Program points, meanwhile, serve as hooks for making assertions about machine state. These are written in separation logic where the keys (the pointers) may be either actual pointers into memory regions or names of machine registers. Additional immutable variables may be defined to assist in the state modeling. Assertions about these are referred to as *pure* or *time-invariant* assertions and are not tied to program points. Procedures additionally have a *linkage*, which defines the external interface to the procedure. In Grouper, linkages can either be C linkages, which define some number of parameters and a return value in accordance with the target platform's C calling conventions, and involve additional blocks generated by Nautilus, or *assembly* linkages, where none of that happens and the interface, if any, must be specified explicitly.

Grouper is divided into two parts: a front end that lowers machine-independent specifications to machine-dependent specifications and a back end that processes block specifications, generates synthesis problems, and produces output programs. The front end of Grouper is closely comparable to Anchor, the Alewife compiler. (It is not the *same* as Anchor because each is specialized to its own specification language, and these specification languages are rather different. Meanwhile, the front end of Grouper is built into Grouper rather than being a separate tool for reasons arising from the history of its design and implementation.)

The back end is responsible for elaborating the Grouper specification and integrating specifications for externally generated blocks, then producing synthesis problems and gathering the synthesis results to combine into output procedures. (These are then output as Tuna files for use by Hermitcrab.)

Each of the block generation tools has a different concept of specification. While Capstan, the synthesis engine, works with Hoare logic specifications similar to Grouper's, the other tools work from less general, tool-specific specifications. Each non-synthesis block has its own specification outside of Grouper. Each block generator produces, in addition to its output assembly text, a Grouper specification for that block in Hoare logic. Grouper reads these and uses them as part of the process of generating synthesis problems. (This is discussed further in Section 7.4.) Note that the non-synthesis tools *produce* specifications and synthesis *consumes* specifications. This asymmetry makes synthesis blocks and the blocks generated by the other tools fundamentally different.

It is also possible that some tools also need or want input from Grouper; in particular, Grayling allows the choice of registers used for inputs and outputs to its blocks to be externally determined, and Grouper provides these. This, combined with a desire to have all the tools run "offline" for debuggability, gives rise to a three-phase execution model for the Grouper backend. All phases read the Grouper specification and (deterministically) lower it to a machine-dependent specification. (Section 7.3.)

- The "prepare" phase reads the block specifications from all tools not requiring input from Grouper, and produces the inputs from Grouper for all tools that do. (Other than Capstan, the synthesis engine.) This currently means the location files for Grayling.

- The "generate" phase reads the specifications produced by all the external tools (besides Capstan) and generates synthesis problems. These synthesis problems are machine-dependent specifications expressed in Cassiopea. (Not Alewife.)

- The "collect" phase reads all the output assembly code, however produced, and collects it into complete procedures suitable for use in (or as) assembly source files.

Nautilus runs before the prepare phase; Grayling runs between the prepare and generate phases; Capstan runs between the generate and collect phases. To intersperse these runs without running any of the tools “online” from inside Grouper (which would significantly impede visibility into the system and thus debugging) it is important that Grouper’s specification processing be deterministic. Note however that the sensitivity of the output to failures of determinism is limited: the collect phase is sensitive only to the number and names of blocks, and while it is perhaps possible for the register choices made during the prepare phase to differ in the generate phase, the full block specifications produced by Grayling will force the ultimate output to be correct regardless. (This is discussed in more detail in Section 7.4.1.)

The remainder of this chapter discusses first the Grouper specification language (Section 7.2), then the front-end specification lowering (Section 7.3), the specification elaboration done in the back-end (Section 7.4), the three backend phases and issues pertaining to each (Section 7.5), some additional considerations (Section 7.6), and finally concludes with a discussion of future Grouper work in Section 7.7.

7.2 Grouper Specifications

As discussed in the previous section, there are two levels of Grouper specifications: the original specification as written by the OS architect, which is machine-independent, and the lowering or specialization of this to a machine-dependent specification for a particular target machine architecture. This section discusses both of them; the next section covers the issues and challenges in getting from one to the other. (The sections beyond all work with the machine-dependent representation.)

7.2.1 Machine-Independent Level

Grouper specifications define *procedures* (standalone, callable units of code) in terms of *blocks* (small, potentially synthesizable units of code). The concrete Grouper syntax allows multiple procedures (that can share declarations) per input file; however, each procedure is treated as an independent unit and processed as if it privately contained its own copy of the shared declarations. For this reason I shall ignore these details for the remainder of this chapter and concentrate solely on procedures.

```
proc idle {
   import {
        insn_idle: block from asmdefs.tuna in machinedir;
        irqon_during_idle: bool from asmdefs.tuna in machinedir;
        irqon_after_idle: bool from asmdefs.tuna in machinedir;
        # get irqon
        ** from controlregs.tuna in machinedir;

        ** from regs.tuna in machinedir;
        misc_scratch_reg: 32 bit loc from hacks.tuna in machinedir;
   }

   var readystate: 1 bit;
   var afterstate: 1 bit;

   linkage: name = "cpu_idle"  globl = true;
   body {
        pre
        { synth: hint scratch misc_scratch_reg }
        ready
        { literal: insn_idle }
        _
        { synth: hint scratch misc_scratch_reg }
        after
        { synth: hint scratch misc_scratch_reg }
        post
   }

   assert {
        readystate == (if irqon_during_idle then 0b1 else 0b0);
        afterstate == (if irqon_after_idle then 0b1 else readystate);

        pre |- [irqon -> 0b0] * true;
        ready |- [irqon -> readystate] * true;
        after |- [irqon -> afterstate] * true;
        post |- [irqon -> 0b0] * true;
   }
}
```

Figure 7.1: *Example Grouper specification. This is for the Goldfish procedure* `cpu_idle`*, described in Section 8.3.4. It uses a literal block for the idle instruction itself, which is taken from the machine description Tuna files, and synthesizes three blocks that potentially manipulate the interrupt state based on the machine's requirements. The* `irqon` *register, which is one bit wide, might be either a physical register in the machine (as on MIPS or x86) or a virtual one computed from others (as on ARM).*

A Grouper procedure has four parts: declarations and imports, a *linkage*, a *body*, and assertions. Declarations may be of types, registers, variables, memory regions, and (to a limited extent) functions. Variables are immutable and may be of any type, not just machine-level types. Functions are not allowed to be recursive. The linkage defines the external interface to the procedure. The body gives the structure of the procedure, in terms of blocks (which may be thought of as statements) separated by program points, and the assertions provide the specification for the body. The complete abstract syntax is given in Appendix B. An example appears in Figure 7.1. A summary of the syntax follows.

Types t include bitvectors, registers, and sets of registers, all of some bit width w, which can be implicit in most contexts. Implicit widths are resolved by unification while typechecking. Types also include booleans, strings (both single-line and multi-line strings like in Tuna), user-defined monomorphic sums and products, a special-cased polymorphic `option`, user-defined structures/records, functions, and alias names for other types.

Declarations d may introduce sum and structure types, alias names for types, registers, variables, memory regions (Section 7.2.3), and (to a limited extent) functions. Format strings can be imported from Tuna, which means that they have function type and take string arguments. Finite maps from registers can be imported as well; they have function type and take register arguments. Variables, such as `readystate` in Figure 7.1, are declared with types, but not values; values are given to variables only by assertions. (This is necessary when those assertions are separation assertions.)

Declarations (and assertions) may also be wrapped in declaration-level match constructs, which unpack sum and product types and contain more declarations. This allows customizing the specification based on imported machine-dependent declarations. It also complicates typechecking beyond the point where it is worthwhile; the specification is only typechecked after all matches have been simplified.

An import specifier names a directory and a file, and either imports a single symbol or all symbols from the file, which is ordinarily a Tuna file. The directory is a symbolic name rather than a literal; the actual locations for such symbolic names are defined on the Grouper command line. There are three symbolic directories by default and the user can define more. The standard directories are:

- `specdir` – the specification file's directory
- `machinedir` – machine description directory
- `outputdir` – the directory for output files

This gives a model where the machine-independent specifications are read from one directory, the machine-dependent definitions another, and files generated by other tools or prior runs from a third, in which Grouper also places its output files. This level of indirection has been useful for managing large systems of specifications, such as found in Goldfish.

The linkage may be either an assembly linkage or a C linkage. In an assembly linkage, control begins immediately at the first program point (and returns immediately after the last program point) and any interfacing must be done explicitly in the specification. The assembly linkage definition allows setting basic properties of the procedure: its assembler/linker-level name, and whether to mark it as global or not. In a C linkage, however, the procedure is assumed to take some number of arguments and return some value in accordance with the usual C calling conventions for the target machine. This is accomplished by referring to a Nautilus specification for the procedure; Nautilus generates any needed code to establish and clean up the stack. Nautilus produces both output blocks with the code and specifications for those blocks; the specifications tell Grouper where the argument values are. These blocks are processed during the integration step (Section 7.4.1). The Nautilus reference in the Grouper specification is essentially the same as in a procedure call block, below, though of course the external Nautilus specification must be different.

The body is a sequence of block specifications Program points x_{pp} are labeled explicitly, before and after each block. Only one program point appears in any given place; that is, the post-point of block A is the same as the pre-point of block B following it. Blocks preserve values by default; that is, block specifications must specify all changes to machine state, but need not enumerate all the unchanged state elements.

There are currently four kinds of blocks:

- Literal blocks
- Synthesis blocks from Capstan
- Context blocks from Grayling
- Procedure call blocks from Nautilus

Literal blocks are just that: a multi-line string copied directly into the output assembly text. Normally the string itself is taken from a machine-dependent Tuna file. (It might also be a format string, with arguments from the specification inserted into it.) This is used when the best way to pick an instruction is to have the porter provide it directly, such as for memory barriers.

Each of the other block types requires specifying assorted properties; these are part of the body specification. For synthesis blocks this is limited to hints providing scratch registers. For context blocks, the following must be provided:

- The name of the Grayling specification file.
- The symbol name in the Grayling specification.
- A register for each of the inputs and each of the outputs of the Grayling block.

The name of the specification file is used to find the Grayling output when integrating. The registers are passed to Grayling by the "prepare" phase.

For procedure call blocks (and C linkages) the following must be provided:

- The name of the Nautilus specification file.
- The symbol name in the Nautilus specification.
- A bitvector value (not a register) for each of the arguments/parameters.
- A bitvector value for the return value.

For C linkages a value can also be attached to the return address and stack pointer, optionally different for entry and exit (this is used by `longjmp`). Also, ordinarily the C linkage automatically asserts that callee-save registers are preserved by the procedure. A flag is available to disable this, since one of the reasons for writing a procedure in assembly language is the ability to do something other than the usual things with callee-save registers. Both `longjmp` and thread switch make use of this flag.

A call block functions as an assertion about the values given for arguments and the return value: the block precondition asserts that the values given in the Grouper specification are in the registers named by the block specification produced by Nautilus, and the postcondition asserts that the return value given in the Grouper specification is in the return value register. This turns out to be very elegant.

For many specifications all register usage is implicit: the calling conventions known to Nautilus determine register bindings for the values, Grouper learns this when it integrates the Nautilus specification, and then the value propagation pass carries these register choices through. This considerably reduces (relative to Alewife specifications) the amount of material that the porter must specify by hand. Intended future work to support register allocation in Grouper (Section 7.7) should substantially reduce the number of specifications that need explicit register management. (Explicit register management must, by necessity, be different on every machine, so it increases the porting burden.) It is possible that using values rather than registers to interface to context blocks would also improve specifications; this is a matter for future investigation.

Note that the specification produced by Nautilus also must, in its postcondition, explicitly apply a fresh value for every call-destroyed register. The default assumption for all blocks is that registers (and memory slots) not explicitly changed by the block's postcondition are preserved. (Due to the way Nautilus is implemented, this material is added by Grouper when it integrates Nautilus specifications.)

Expressions e include, most notably, a form for pointers and a match construct for unpacking the sum and product types. Pointers have the form $[x_m, e]$, where x_m is the name of a declared memory region. Memory regions are not first class. (See Section 7.2.3 for a discussion of the pointer model.) Identifiers declared as registers (written x_r) are treated as values and are distinct from variables of register type. Note that this is different from Cassiopea; it makes handling the separation logic easier.

Assertions a may be expressions (which apply to variables independent of machine state) or assertions in separation logic about the machine state, attached to a program point x_{pp}. In Figure 7.1 there is a separation assertion about the control register `irqon` for each of the four named program points. Separation assertions have the usual forms for separation logic, with keys that can be either registers or pointers, plus an additional nonstandard construct $[x \leftarrow e]$. This is pronounced "find e" and asserts that the variable x is equal to the (or a) location holding the value of e. (Thus, $[x \leftarrow e] * [x_r \to e]$ reduces to $x = x_r \wedge [x_r \to e]$.) This is useful for extracting the identity of registers used implicitly in a specification and discussed further in Section 7.4.2.

Observe that the assertions (even those attached to program points) are separated from the body, both in the concrete syntax and in the internal representation. This was an intentional design decision for the concrete syntax, based on early experience writing specifications (in a rather different form at the time) and also on past observation. There is a tendency, when working with Hoare logic, for the assertions to be larger than the statements they surround and to therefore quite effectively obscure the structure and flow of the code. Separating the assertions from the body in the concrete syntax was intended to eliminate this problem: the body declaration would give a clear overview of the procedure's structure, and the assertions afterward would refer to points in it, much like footnotes. This was also partly motivated by the separation in Coq between code and theorems about code, which I have felt is one of Coq's strengths. Unfortunately, in the case of Grouper, my conclusion is that this distinction was a mistake: because of the elaborations of the specification, particularly value propagation (Section 7.4.2), the assertions attached to program points are rarely large or cumbersome.

```
proc idle {
   type word = 32 bit;
   register cp0_12_ie: 1 bit = "<wrong[ie]>";
   register v0: 32 bit = "$2";

   linkage: name = "cpu_idle"  globl = true  leaf = true;
   body {
      pre
      { synth: hint scratch v0 }
      ready
      {
         literal: <"
               .set push
               .set mips32
               wait
               .set pop
            ">
      }
      .pp0
      { synth: hint scratch v0 }
      after
      { synth: hint scratch v0 }
      post
   }

   assert pre |- [cp0_12_ie -> 0b0] * true;
   assert ready |- [cp0_12_ie -> 0b0] * true;
   assert after |- [cp0_12_ie -> 0b1] * true;
   assert post |- [cp0_12_ie -> 0b0] * true;
}
```

Figure 7.2: *Example lowered Grouper specification; the specification from Figure 7.1 specialized for MIPS. The definitions for the various variables have been imported and substituted; for example,* `misc_scratch_reg` *has been replaced with the* `v0` *register, the* `irqon` *register has been replaced with* `cp0_12_ie` *(the interrupt enable bit of control register 12) and* `readystate` *has been replaced with zero. The assembler string for* `cp0_12_ie` *is given as* `wrong` *because it is part of a control register and thus not directly accessible from the assembler. Observe that the block between* `pre` *and* `ready` *is degenerate: nothing changes so nothing needs to be done.*

Note: this example is pseudo-syntax, hand-edited from a state dump, and may consequently contain mistakes.

Simultaneously, unlike when using Hoare logic for verification, the body declaration of a Grouper procedure is not standalone and cannot not really be read to see what the procedure, or any step in it either, is supposed to do. This negative result is disappointing, but probably does not generalize to verification settings. (The separation in the internal representation reflects the concrete syntax. The assertions are eventually collected into the body, but only well into the backend processing. See Section 7.4.)

7.2.2 Machine-Dependent Level

After the specification has been specialized, it is lowered to a second internal representation. This representation is essentially the same as the representation described above, except it lacks the following elements:

- Alias names for types.
- All sum types and match constructs.
- Imports.
- Unapplied format strings.

Additionally, some properties that are expressions in the machine-independent representation become constants. In short, everything that is conditionalized on the machine type in the machine-independent specification has been resolved and simplified. A lowered version of the Figure 7.1 example is shown in Figure 7.2.

After integration (Section 7.4.1), the representation is transformed again, this time to one where the separation logic assertions appear as finite maps from the key space to the value space rather than as syntax trees. (The key space is also accordingly restricted to register names and pointers, and the value space to only variables and constants.)

7.2.3 Memory in Grouper

Grouper treats memory as a collection of small disjoint blocks, not as a whole address space, because it works with a programmer's view of the machine. Memory regions are symbolic; their actual addresses are not known. Furthermore, pointers may not be created out of thin air; they must be either attached to linker symbols or passed in as arguments from machine-independent kernel code, which might get them from `malloc`. (In the case of linker symbols the addresses are constants and must be loaded into registers for use; but as one would expect, the values for the constants are assigned by the linker and not known at specification processing or synthesis time.)

Blocks of memory are weakly typed, in the sense that they are handled as collections of machine integers located at offsets from the base address rather than being just collections of bytes. A block of memory can either be an array (k elements of the same machine integer type) or a structure (potentially heterogeneous elements at arbitrary but constant offsets specified by a structure type) but not both. (Arrays of structures, and structures containing arrays, are both perfectly valid concepts, but neither is required for the code we need to generate.) There is also no way to subdivide memory regions; this has not been necessary.

Accesses to undefined or out of range offsets are not permitted; writing them in specifications will cause errors. Furthermore each element can be accessed only as a machine integer of its declared type, not a smaller one. This rules out a few assembler tricks (such as truncating a value to 8 bits by using an 8-bit-wide instruction to fetch it from memory), but as discussed in Chapter 5, these tricks can be recovered by applying optimizer passes after synthesis.

Pointers are defined as pairs of a memory block name and a (machine) integer offset. Offset zero is the base address of the block. Addition and subtraction on pointer values updates the offset, but cannot change the block name. Thus for example it is not possible to add some offset to a pointer to one block to get a pointer to another block (even though in the underlying machine, some such offset exists) and two pointers to different blocks are never equal. (A specification that asserts such an equality is unsatisfiable and will be rejected with an error.)

7.2.4 Concurrency in Grouper

Memory access brings up another topic: concurrency. Operating system code is inherently concurrent, and the code specified Grouper is no exception. However, almost all operating system code, whether machine-independent or machine-dependent, is not itself *strongly* concurrent in the sense of performing machine-level operations that require reasoning about concurrency; rather, it is written using *concurrency primitives* that provide well-understood (and normally machine-independent) guarantees to the code around them. In particular, most kernel code is written using locks in such a fashion that explicit concurrent reasoning is not required for the code within the critical sections protected by the locks. The chief exceptions are (1) small amounts of code deliberately written otherwise for performance and scalability reasons, which in this project we avoid for pragmatic reasons; and (2) the implementations of the basic concurrency primitives themselves, which as we saw in Chapters 4 and 5 are (for the time being) best hand-written by the porter. Both these restrictions are amenable to being lifted by future work; see Chapter 9.

As a result, Grouper itself need not support reasoning about concurrency. Procedures generated via Grouper can, if necessary, use the OS's concurrency primitives directly by calling the appropriate functions; Grouper cannot itself reason about this, but most kernel programmers do not expect such support. That the specifications for such blocks need only be able to prevent Grouper from making incorrect assumptions, and as we will see, that is possible.

7.3 Lowering Specifications

Grouper procedure specifications must be machine independent. This requires all references to machine-dependent phenomena (for example, the number of registers and their names and sizes) to be abstracted. Grouper uses two complementary techniques to achieve this: *refinement* and *predicate abstraction*. Both of these are achieved by substituting machine-dependent values imported from Tuna files.

7.3.1 Refinement

Refinement broadly means transitioning from an initial specification to a more detailed version that unrolls abstractions or introduces additional complicating details. In Grouper, refinement introduces a partially machine-dependent concept or construct to allow writing a specification (or part of a specification) that relies on that construct. Doing this unconditionally is pointless, so each refinement is a choice among at least two options, where each option reflects a different possible approach to the problem suitable for a different set of machines. For example, while in a uniprocessor kernel the identity of the current thread or current process can simply be kept in a global variable, in a multiprocessor kernel every CPU has its own current thread, so there is some sort of per-CPU data structure, and it stores the current thread. To find this efficiently (since it is frequently referenced) each CPU stores something on-chip that lets it get to the data. One way to do this is to reserve a register that always (at least when executing in the kernel) holds the current thread pointer. One can instead reserve a register for the current CPU pointer. Another way is to install a per-CPU memory mapping so that the CPU structure for the current CPU can always be found at a well-known address.

Some machines have extra registers not accessible in user mode in which the kernel can keep such metadata; for these, holding the current thread pointer in one of them is usually the best approach. Others make it easy to install per-CPU memory mappings, and for these the memory mapping method is preferable.

Ideally for each specification refinement the options are in some sense a covering set, such that for every machine of interest at least one of the choices is suitable or at least feasible. It is not entirely unreasonable to think that one can construct such covering sets.

However, this raises an important point: operating systems, or at least those that have been ported, already come with models of how machines vary. Off-the-shelf OS/161 already (since 2009, when it first got multiprocessor support) explicitly understands the distinction between holding the current thread pointer and holding the current CPU pointer. The choice affects the order of some initializations in the machine-independent thread code. Because that support is there I can write a set of refinement choices for handling the current thread that use both approaches and have them work. Otherwise, I could write the general set of refinement choices, but I would also have to generalize the model reflected in the machine-independent code. (And since the order dependence in question is not particularly obvious to casual inspection, I would likely have to discover it by debugging crashes during kernel initialization.) This would be necessary regardless of whether the machine-dependent code was specified in Grouper or whether I was doing a traditional port and writing code by hand.

So while it is difficult to guarantee that one really has a covering set of alternatives, and this can be seen as a weakness of the refinement technique, it is important to realize that this is not a new problem. The generality of the models already designed and built into operating systems (especially relative to future hardware yet to be invented) is a long-term system design problem for which no easy solutions exist. We will see in Chapter 8 that existing standard practice is, for the most part, vastly overgeneralized. Meanwhile the track record of OSes at adapting easily to new hardware functionality is, at best, mixed.

Models written as explicit sets of refinement alternatives in specifications are clearly superior to models that exist only implicitly as emergent properties of large bodies of code. This is true both in general and when one discovers one must add an additional case. It is even more true in the long run when we can hope that the machine-independent code will have been verified against all the existing cases of the refinements.

In Grouper specifications, refinements are (in general) implemented using match constructions on sum types. This works very nicely: each constructor of the sum represents a different choice, and the arguments of the constructors, where needed, carry any parameters or other information specific to that choice. (For example, in the case of storing the current thread in a register, which register to use would be an argument of the constructor.)

As described in the previous section Grouper supports not just an ordinary expression-level match but also a declaration-level match; this gives a great deal of descriptive power at the cost of serious complications for analysis of the specification. I do not attempt to typecheck specifications until the declaration matches (and in fact all matches) have been substituted and resolved.

7.3.2 Predicate Abstraction

The other technique used to make specifications machine-independent is *predicate abstraction*. This is the practice of writing specifications using abstract elements (particularly predicates, but also e.g. variables and registers) that are then concretized on a per-machine basis. For example, one might write the specification for a block to turn interrupts off as follows:

```
pre: interrupts_on() == true
post: interrupts_on() == false
```

and then define `interrupts_on` as needed for each machine – e.g. on x86 it would read the status word bit controlled by the `sti` and `cli` instructions, on MIPS it would read bit 0 of coprocessor 0 register 12, on ARM there are two bits to OR together, etc.

This is a general technique, in the sense that one can, if necessary, use it to write any specification at all by putting all the logic in the concretizations. However, it is also a rather blunt hammer in that doing so sacrifices any real concept of machine-independence in that specification. It has seemed to me that there is not always an entirely satisfactory intermediate point where the machine-independent logic stays mostly in the machine-independent specification and the machine-dependent logic remains in the concretization definitions. (Or, perhaps, such intermediate points in general exist but are hard or not obvious to derive.)

As mentioned in Chapter 5, Alewife, the machine-independent specification frontend language for our synthesis engine, is based on predicate abstraction. The original intention with Grouper was to have it generate still partly machine-independent specifications in Alewife and use the Alewife compiler's predicate abstraction to concretize them. I found that refinement was an effective way to write the top-level parts of many of the specifications I was encountering: it most naturally applies when a problem can be approached several different ways, such that the specifications of the different ways are still largely machine-independent. Consequently, the functionality needed for refinement was intentionally removed from Alewife, and Grouper was initially designed with no predicate abstraction.

This model broke at two points. First, Grouper reasons about register usage, so it needs to know about registers and they need to be concrete when it does so, but all registers in specifications must be abstracted so as to be machine-independent. This means that Grouper must be able to concretize abstract values. (Its ability to do so is somewhat more limited than Alewife's is, but only because neither Grouper's specification language nor the Tuna interchange language supports arbitrary functions.)

Second, because Grouper uses separation logic to reason about registers and other machine state, it is unsafe for Grouper specifications to use abstract predicates that are concretized by the Alewife compiler to include references to machine state. The `interrupts_on` example above would be safe, because it refers to control registers that the Grouper specification itself does not touch (given a way to crosscheck that). However, other predicates one might want to use, for example to check the return value from a procedure, are unsafe. Unfortunately, Alewife does not, in general, support *state* abstraction; one can declare an abstract register and concretize it with a real register, but one cannot concretize it with a subfield of a real register. Nor can one concretize it with two real registers pasted together. This means that `interrupts_on` cannot be written in terms of an abstract Alewife state that can appear in Grouper's separation logic.

These problems led me to improve Grouper's ability to concretize abstractions to handle predicate abstraction itself and abandon the use of Alewife. Grouper now has, as described already, a front end that produces specialized machine-dependent specifications and a back end that processes them to output machine-dependent block specifications for synthesis directly to the Capstan synthesis engine.

For state abstraction, however, I took a slightly different course. Most state in most specifications involves general-purpose registers and can be handled by substituting registers, which is straightforward. Other state cannot be handled that way; for example, on many machines (such as ARM and RISC-V) the interrupt state is the combination of two or more bits, potentially in different control registers. Clearly the desired way to work with this state in a machine-independent specification is by treating it as a virtual register and writing separation logic assertions about it like any other register. However, expanding such a virtual register into those assertions is not entirely trivial.

Consider the fragment $x_{pp} \vdash [\texttt{irqstate} \rightarrow \texttt{0b1}]$. This should be read "at program point x_{pp} the virtual interrupt state is 1". Suppose for the moment (this example is not from any real machine) that the interrupt state is the AND of bit 0 of register ctl0 and bit 4 of register ctl1, both registers being 32 bits wide. The expansion of this assertion requires reading both control registers, projecting the bits from them, and combining the results. This can be written as follows:

```
var val0 : 32 bit
var val1 : 32 bit
var bit0 : 1 bit
var bit1 : 1 bit
var irqstate : 1 bit
assert bit0 = narrow val0 1
assert bit1 = narrow (val1 >> 4) 1
assert irqstate = bit0 & bit1
assert irqstate = 0b1
assert $x_{pp} \vdash [\texttt{ctl0} \rightarrow \texttt{val0}] * [\texttt{ctl1} \rightarrow \texttt{val1}]$
```

(It could also be written with fewer variables at the expense of clarity.) Both the machinery to declare this substitution and to produce it would be expensive. Moreover, the substitution must be done cautiously: naive introduction of whole control registers into a separation assertion referring to portions of them can falsify the assertion. Each substitution register must be introduced exactly once, no matter how many pieces from it are being referenced, and the pieces must be explicitly checked for overlap.

As a matter of expedience I decided to avoid doing this transformation at all. The machine-dependent specifications Capstan accepts are written in a pure dialect of Cassiopea, and can take advantage of many Cassiopea language features, in particular its functions. Grayling already has a feature that lets it emit register accesses to Capstan as function calls. (This was necessary to allow it to work with control registers; see Chapter 6). I made virtual registers in Grouper work similarly:

accesses to them are emitted as function calls. Then I can define the supporting functions in each Cassiopea machine description. This allows replacing all the expensive machinery with a single flag attached to each Grouper register. The drawback is that this mechanism is not fully safe: there is currently no way to assert in a Cassiopea specification that the read footprint of two functions is disjoint, or does not include some other register. However, the nature of the procedures involved (and the small number of virtual registers needed) makes this a minimal risk: for example, the procedures in Goldfish that turn interrupts on and off do *exactly* that and touch no other control register state. Adding a way to crosscheck disjointness of function-based registers to Cassiopea and Capstan would be fairly straightforward future work.

7.3.3 Lowering Implementation

For both refinement and predicate abstraction the underlying implementation method is to import definitions from per-machine Tuna files, then substitute them. For this reason, in the Grouper frontend, after importing everything and typechecking (and after an alpha-conversion for simplicity downstream), there is an aggressive substitution and simplification pass. Any time-invariant assertion that has a variable by itself on one side of an equality is used as a substitution. Assertions that contain variables by themselves on both sides of an equality are used to collect variables into equivalence classes, and only one representative from each class is retained. Constant folding then allows removal of (in particular) match constructs. Specifications where this pass fails to eliminate matches or other elements not permitted downstream (see Section 7.2.2) are rejected.

7.3.4 Retargeting Grouper

Grouper itself needs to know some things about the target machine. It reads these things from a Tuna file called `grouperdefs.tuna`. Most of these items are output strings for producing proper assembly language for the target machine, such as the preamble that goes before each procedure. The others are sizes of things (such as the bit size of pointers) and the sets of caller-save and callee-save registers. One additional item is that machine invariants declared in the Cassiopea machine description (such as an always-zero register being zero) are not handled by Capstan (the synthesis engine) directly; rather they are inserted into specifications by the Alewife compiler. It is therefore necessary for Grouper to also insert these assertions as well.

7.4 Elaborating Specifications

After lowering to a machine-dependent specification, Grouper performs two elaboration passes on the specification. The first (integration) loads the specifications for external non-synthesis blocks; this is of course critical to interoperating with multiple block generators. The second (value propagation) applies the principle of inertia: any value in a register or memory cell remains there at the next program point unless explicitly overwritten. This turns out to be helpful for both making specifications more concise and allowing many machine-dependent details to be implicit.

7.4.1 Integration

Integration is the process of reading material produced by external generator tools (other than the synthesis engine Capstan) and merging it into the Grouper specification. In the case of blocks generated by both Nautilus and Grayling, and any future tools that produce rather than consume specifications, this means replacing the external block reference in the internal representation with a literal block (containing the externally generated code) and merging the externally generated specification into the assertions. (And a C linkage is replaced with literal blocks and assertions about an assembly-level linkage.)

When an external block and specification are merged, Grouper creates two new program points around it and two additional synthesis blocks between these and its previously adjacent blocks. That is, the sequence

```
before
{ call foo() }
after
```

becomes

```
before
{ synth }
before_call
{ literal ... }
after_call
{ synth }
after
```

This serves two purposes: it avoids any need to merge the external assertions with any user-provided assertions attached to the pre-existing program points, and it provides a block in which, if necessary, values in user-selected locations can be moved to the locations demanded by the external block. This is not normally an issue for Grayling blocks, where the locations are selected explicitly, but sometimes can be for Nautilus blocks, where the locations are dictated by the target machine's calling conventions. It also turns out to be important for the overall soundness of Grouper: naively merging different sets of assertions about the same program point can produce an unsound specification. Consider as an example two adjacent call blocks: the specification for the first asserts the machine state it produces, and the specification for the second asserts the machine state it expects. Merging these assertions safely at the single program point between them would require that these machine states match exactly, which in general they will not. Two distinct program points are required, so at least one block must be inserted. Often the inserted blocks do not need to do anything so their specifications become degenerate. This is harmless: while synthesizing large blocks can be slow, synthesizing *empty* blocks takes only a fraction of a second. (One could easily have Grouper automatically detect empty blocks and generate the empty code block itself. However, it turns out to be important for build/makefile reasons for the number and names of output block specifications generated by Grouper to be predictable.)

For ordinary blocks, the inserted blocks surround the external block. For linkages (C linkages only, since assembly linkages have no implicit behavior) the inserted blocks come between the generated prologue block and the first program point of the specification, and between the last program point of the specification and the generated epilogue block. Notice that this means that for a procedure with C linkage the user cannot write assertions about the machine state on entry to the procedure. Assertions attached to the first program point of the specification will come after both the prologue and its inserted block; rather than functioning as preconditions for the procedure they function as conditions the inserted block should try to establish based on the state created by the prologue. This is intentional.

Specifications of the C-level interface to procedures should come from C-level declarations. In a future world where the machine-independent kernel was verified, a less hackish version of Nautilus would, in addition to compiling the C-level function signature into an assembly-level linkage specification, compile the C-level program logic into an assembly-level specification that would dovetail with Grouper's own program logic. Likewise, for calls it would compile the callee's C-level program specification into an assembly-level one that Grouper can work with. This project is not yet in that world. Mostly the C-level specifications go unstated (this is occasionally convenient when they involve awkward extra complications). To work around the occasional need for such specification material, I added an extra attachment to C linkages to allow injecting extra assertions into the real start and end program points. These must separate from (that is, not involve the same registers as) the normal C linkage logic, or Grouper will reject the specification.

The external specifications generated by Nautilus are rather stylized: there are two assertions, one stating that at the `pre` program point various registers point to dummy variables `arg0`, `arg1`, etc. and at the `post` program point some register points to another dummy variable `ret`. For linkages dummy variables `retaddr` and `stackpointer` are also bound at both program points.

For calls, Grouper replaces the dummy argument variables with the real argument and return values provided in the call block reference; for linkages, it replaces them with the parameter and return values provided in the linkage definition. In both cases it renames the program points `pre` and `post` to the newly generated program points associated with the inserted synthesis block.

Grouper also attends to callee-save and call-destroyed registers. (Properly speaking this should be Nautilus's responsibility, but because Nautilus is a hack designed to avoid needing real knowledge of calling conventions, it has no such knowledge. Meanwhile, Grouper already needs to know about registers, so it is expedient to have Grouper handle this issue.) For call blocks, Grouper marks each call-destroyed register destroyed at the post-point of the call, by asserting it points to a value given by a fresh variable. These variables have reserved names and are not otherwise accessible, so function as arbitrary indeterminate values. For linkages, conversely, Grouper generates a variable to refer to the initial value in each callee-save register and asserts that it holds this value at both the entry and exit program points. As already mentioned, for some procedures this latter behavior needs to be circumvented. One of the reasons some procedures are traditionally written in assembly language is that they do unconventional things with the callee-save registers. In particular, the C library function `longjmp` does *not* preserve them; it restores the callee-save register values from an earlier point before returning there. Similarly, thread switch restores a different thread's callee-save register values. Both of these violate the default assertions. Consequently I provide a setting to turn them off.

For Grayling blocks the integration mechanism is somewhat more complicated. Grayling produces real, complete, and verifiable specifications for its output blocks. (See Section 6.9.) These can be imported directly, with only the program points needing to be renamed. However, Grayling generates assertions about four sets of program points: two for the points immediately around the generated block, and two more to be placed at the beginning and end of the procedure. (Or, optionally, elsewhere.) The latter two assert that the values to be saved from registers (or restored to registers) are not destroyed by other blocks before being saved (or after being restored), and that the values in memory after being saved (or before being restored) are not changed. For exception handling, these assertions must be anchored not at the beginning and end of the procedure, but rather must meet in the middle so that the saved values from the exception entry become the restored values for the exception return. Additionally, the memory region for the context operation must be known to both Grayling and Grouper. I handle this by giving it a name in the external Grayling specification (since the name can be machine-independent) and then using the same name in the Grouper specification. On integration, if a memory region found in the Grayling specification has the same name as one already present in the Grouper specification, the two must match.

Specifications for external blocks are placed in `.grrbl` ("grouper block") files, one file per block. These have the same syntax as Grouper specifications, except are limited to variable declarations, memory region declarations, and assertions. The generated code is placed in a Tuna file, along with a declaration of the type of the function (for Nautilus blocks) or the types of the inputs and outputs (for Grayling blocks). Grouper uses these to assist the integration, and also to cross-check that the external block reference in the Grouper specification was correct. (For example, it checks the number and types of argument values provided for call blocks.)

Notice that both integration and importing machine-dependent definitions for substitution and lowering read external files, often in the same formats and using some of the same apparatus. Recognizing that these two steps are nonetheless fundamentally different, should happen separately, and in fact belong to different processing stages, was critical for developing a reasonably structured implementation.

7.4.2 Value Propagation

One of the primary motivations for developing Grouper was to avoid writing the same assertions over and over again. Therefore, after the specification and all additional input material available in the current phase is read, a transformation is applied to enable this.

The basic premise is inertia: that given two program points p and q, anything that is true at p should remain true at q unless it has been explicitly invalidated. So for example if at p, a register contains the value x, unless the assertions for q give a different value, at q it will still contain x. This leads to an additional principle for synthesis blocks that the synthesis engine is required to adhere to: it may not destroy registers unless they are either (a) used in the postcondition of the synthesis block, or (b) declared explicitly as destroyable. This creates complications for the synthesis engine, but also helps to make synthesis faster by reducing the program space. See the Cassiopea paper [16] for further discussion.

However, we do not want to assert that the register contains the value x at q if x is dead, that is, the value is never used again. This would prevent being able to reuse the register to hold other values or as a scratch register.

Consequently value propagation is a three-pass transform: first it does a forward pass to determine where, for every value that appears in machine state, that value first becomes available. Then it does a similar backward pass to determine where every value is last used. This gives it a set of live ranges for each value. (Values are Grouper variables or constants. A preprocessing step introduces fresh variables for any more complex expressions that are encountered. This step often ends up undoing substitutions made by the front end during lowering; consequently it must be careful to generate only one variable per distinct expression.)

Then it works forward maintaining a bidirectional mapping between locations (elements of machine state) and values. (Note that the separation logic representation is critical to being able to do this.) When stepping from point p to q, and at p element r contains value x, if the block between p and q does not assign a different value to r, and x remains live at q, it adds to the specification an assertion that, at q, r also contains x. (This is written $q \vdash [r \rightarrow x]$, concatenated to the other register assertions at q with the $*$ separation operator.) If x remains live at q and the block between *does* assign a different value to r, then perhaps at q something else contains x. If so, all is well. If not, it adds the triple (p, r, x) to a list of values that need to be saved. When it next sees x, say at point p', it adds this information as well.

Upon completion value propagation has done two things: (1) it has enriched the assertions with the available additional information about machine state, and (2) produced a list of values that need to be preserved across blocks where currently nothing holds them. This list can then be handed off to a register allocation pass. Register allocation (including spilling values to the stack) remains future work, however, so currently if such values are found, the specification is rejected. In practice this can be worked around by explicitly placing values in registers in the input specification. (This requires that the porter choose each such register explicitly, however, so is not desirable.)

The nonstandard separation form $[y \leftarrow x]$ ("find x") means that the variable y should be bound to a register holding the value x. This allows extracting the identities of registers chosen by calls and linkages, which in turn mostly avoids the need to explicitly choose registers, which in turn reduces the amount of machine-dependent effort needed to produce a port.

When the value propagation pass encounters this form, it looks for registers r containing x. If it finds some, it picks one (arbitrarily) and drops the $[y \leftarrow x]$ assertion. Instead it asserts $y = r$ and, afterwards, replaces all uses of y with r. If it finds none, it carries the nonstandard form forward by the inertia rule until it finds one. If it never finds one, the specification is rejected. (Thus the nonstandard find form does not persist past the value propagation pass, and it does not ultimately affect the soundness of the separation logic.)

As of this writing the find construct works nicely when the substitution is forward-propagating (that is, when the value is already in a specific register when the find construct is encountered) but for backward-propagating substitutions, when the value is in or can be placed in a register but the identity of the register is not discovered until later in the procedure (such as a return value register), it fails to work. So far it remains unclear if this is a bug in the implementation or a defect in the design of the feature.

An additional issue with value propagation is that the liveness analysis does not naturally account for pointers. If a memory region is live at a given program point (precisely what that means remains not entirely clear) then a pointer to it must also remain live, because otherwise accesses to the memory region are impossible. (Pointer values cannot be created arbitrarily.) This is true even if the specification itself never mentions the pointer, and therefore requires extra logic in the liveness analysis. As a conservative approximation, and given the nature of the procedures and blocks Grouper is being used to generate, I have decided that all memory regions will be treated as always being live. This currently necessitates workarounds in some specifications.

The liveness analysis also does not account for indirect references to variables through assertions. (variables that never appear in machine state). This sometimes causes specifications to be rejected. The problem occurs when one variable is computed from the value of another, but at least two blocks separate their explicit live ranges. The liveness analysis cannot detect the transitive dependence and does not realize that one or the other value needs to be live across the blocks separating. The proper solution to this problem is to rearrange the internals so that the analysis of assertions done downstream by the code for the generate phase (described in Section 7.5.2) happens before value propagation. Then the necessary information to trace these dependences will be available. In the meantime, various workarounds are possible, such as moving the computation of the value for the second variable earlier or explicitly mentioning the first variable in a later program point.

7.5 Phases

For implementation reasons Grouper operates in three distinct phases. In the *prepare* phase it processes all the imports, checks the specification, integrates external blocks that do not require information from Grouper, and runs value propagation. Then for each external block that does require information from Grouper it generates the necessary outputs. For example, as discussed in Chapter 6, Grayling typically expects to be told which register holds the pointer to its context block. (“External blocks” mean blocks generated by external tools that *produce* specifications, and not synthesis blocks.) In the *generate* phase it reprocesses all the imports, checks the specification again, integrates all external blocks, reruns value propagation, and produces specifications for all the synthesis blocks. In the *collect* phase it reads in the code produced by all the block generators, splices it together along with the necessary assembly language declarations and framing, and outputs the complete assembly procedure.

This architecture is somewhat unnatural, and I have attempted to avoid baking it into the specification language any more than necessary. It arises from wanting to be able to run Grouper as an offline process (relative to synthesis) rather than having it sending off synthesis problems itself and waiting for results. Keeping it an offline process makes it much easier to debug the output synthesis problems, cache synthesis, debug and test in isolation, and so forth. It would really be better if some of the steps could run incrementally.

As things stand it is important that input at later stages does not invalidate earlier stages, because there is currently no way to go back and redo the earlier stages. I will return to this point in Section 7.6.1 after further discussion of the internals.

7.5.1 Prepare Phase

The prepare phase is responsible for generating the locations (`.locs`) file used by Grayling. After specification processing and value propagation this is straightforward: it can read the register names for inputs and outputs directly out of the context block reference in the internal representation.

7.5.2 Generate Phase

The generate phase produces synthesis problems. For each synthesis block, the assertions about the program point before the block and the program point after the block are collected, translated to Cassiopea, simplified, and sent off to the synthesizer.

Collecting the assertions is the first step. There is a problem associated with doing this, however: first, while the separation assertions associated with a block's program points are readily found, they contain only references to variables (or occasionally constants). One needs the assertions about those variables as well. But it does not work to include *all* time-invariant assertions, because some of them may refer to values that pertain to later in the procedure and have not been computed yet. Therefore one must pick and choose. This immediately raises two questions: first, how to do it, and second, whether doing so is sound.

Grouper's soundness hinges on the proposition that if it divides a procedure into synthesis problems, and the synthesis engine produces outputs that satisfy the problem specifications, then combining the blocks into an output procedure also meets the specification. Grouper does not (for the most part) check assertions or satisfiability of assertions itself. Therefore it seems fairly clear that if the decomposition of a Grouper specification into synthesis problems leaves out any assertions, those assertions may not be true about the combination of results into an output procedure and unsoundness follows.

It is not so clear that the converse is true: that if every assertion is enforced by at least one synthesis block, the combined output meets the original specification. A sketch of a proof follows. Call an assertion *enforceable* in a block if it includes only constants, global variables, and variables whose values are available in the machine state at either the point before the block (the pre-state) or the point after the block (the post-state). Assertions that use only values from the pre-state form the precondition; the others form the postcondition. Proceed by induction on paths through the blocks, where the inductive hypothesis for each block is that the values in the pre-state satisfy the assertions in the precondition. (Coinduction may be necessary for loops.) If all enforceable assertions in a block are fed to the synthesis engine and synthesis succeeds, this produces code that computes values for the post-state that are consistent with all the assertions. This provides the inductive hypothesis for the next block. Therefore, since the first block can take the precondition as a premise, over the whole procedure all assertions included must be true. Therefore, if every assertion is enforced in at least one block, any assignment of values to variables that satisfies each block will also satisfy all assertions in the original Grouper specification. Specifications with an assertion that cannot be enforced in any block must be rejected. (Qualitatively, such specifications have the property that they try to store values somewhere outside of the machine state and bring them back later. Since no known machine can do this, these specifications cannot be permitted.) Note that every assertion is also always enforced in every block where it can be. Also note that inserting additional synthesis blocks where external specifications have been imported (as discussed in Section 7.4.1) is critical for this; otherwise the proof fails.

The previous analysis glosses over a complication: a *ghost* variable (one that never appears in machine state) is never available and therefore assertions about such a variable cannot be enforced. This is excessively restrictive. For ghost variables, Grouper looks for *defining assertions*, assertions where the value for one variable can be determined using the values of others. It checks each assertion for possible definitions, including via certain simple transforms. (For example, it can define either x or y from the other using the assertion $x = y + 1$.) Ghost variables with a defining assertion where the definition is composed entirely of elements available in a particular block are also available in that block. (It is unsound to use defining assertions for non-ghost variables.)

An additional safety restriction on ghost variables is needed, however. Consider the following example:

$$\texttt{assert pre} \vdash [\texttt{r0} \to \texttt{x}]$$

$$\texttt{assert mid} \vdash [\texttt{r0} \to \texttt{6}]$$

$$\texttt{assert post} \vdash [\texttt{r0} \to \texttt{z}]$$

$$\texttt{assert}\ (\texttt{a} = \texttt{x} + \texttt{3})$$

$$\texttt{assert}\ (\texttt{a} = \texttt{z} + \texttt{4})$$

There are two defining assertions for a here, one available at `pre` in terms of x and one at `post` in terms of z. Naively saying that these assertions are enforced by including them in the first and second blocks respectively will result in bad code, because nothing has enforced the consistency of the two definitions.

A *sufficient* safety condition for ghost variables is that every ghost variable must be available in a contiguous series of blocks. Then consistency is clearly enforced. This is not a necessary condition (it is also obvious that consistency is enforced if only one defining assertion for each ghost variable exists, regardless of where this assertion is available) but it seems to be adequate for most usage. Specifications that do not meet this condition are rejected. This does not seem to arise often in practice.

Note that passing the above tests does not mean that a specification is valid; it just means that the responsibility for checking validity has been devolved correctly to the synthesis engine, which may find other reasons to object.

An additional comparatively minor complication translating to Cassiopea arises because, unlike Grouper, Cassiopea demands that all variables be defined, not merely asserted about. Fortunately the mechanism for finding defining assertions takes care of this problem. (Specifications where a defining assertion is needed for Cassiopea output and none is found may be rejected.)

It has also proven important to simplify the Cassiopea specifications before printing them, both for readability when debugging and to avoid unnecessary complexity that slows down synthesis. The important steps taken are:

- Before converting to Cassiopea expressions (which do not use separation logic) remove from the postcondition any machine state assertion that is the same as in the precondition. Because unaltered state is invariant, these assertions do not need to be explicit.

- Simplify tautological comparisons. These seem to appear frequently as a result of the integration step.
- Remove unused variables. After the previous two steps, many variables are no longer needed. This can result in fewer registers being mentioned in the specification at all, which in general accelerates synthesis.

7.5.3 Collect Phase

The collect phase is relatively simple: it steps over the Grouper specification collecting assembly text, and for synthesis blocks reads the output produced by Capstan. It appends a return instruction to the end of the collected procedure, inserts an assembler label at the beginning, and wraps the whole in whatever preamble and post-amble have been defined for the target machine's assembly language. (These often include setting the symbol type and size with assembler pseudo-ops.)

7.6 Discussion

This section discusses some additional points of note.

7.6.1 Phase Sequencing

Given Grouper's three phases (prepare, generate, and collect), it is important that input at later stages does not invalidate work earlier, because there is no way to retroactively change the output of a previous phase once it has been consumed by another tool. There are also only limited possibilities for cross-checking it.

There are three ways this can manifest: first, input available only in the generate phase might invalidate the results of the prepare phase; second, input available only in the collect phase might invalidate the results of the prepare phase; and finally, input available only in the collect phase might invalidate the results of the generate phase.

Since Nautilus does not require (or accept) input from Grouper, its output is fixed and available in all stages. This is not true of Grayling, which runs between the prepare phase and the generate phase. The specification material read from Grayling in the generate phase has two parts: pre- and postconditions that specify exactly what each context block does and material that leads to assertions that assure that the input state of registers it saves (or the output state of registers it loads) is not destroyed. This material is partly determined by the location information produced by the prepare phase, and modulo possible bugs in Grayling is constructed to be consistent with it. If there are multiple context blocks in one Grouper procedure, it is possible for one to interfere with another in limited ways (for example, if there are two save blocks, and a register overlapping the second is used as a location input for the first), but this will result in the Grouper specification being unsatisfiable and is an error either in the specification or the concretizations for it written by the porter. Ordinarily however Grouper procedures will contain either only a single context block or a save block followed by a load block where the opportunities for such interference are negligible.

The other cases, where input read only in the collect phase invalidates previous work, requires a bit of explanation. Currently, the generate phase produces synthesis problems, and the collect phase reads the results, which are code blocks that meet their specifications, unless some of the synthesis fails, in which case the whole procedure generation fails. There is no way for this to invalidate work from prior phases.

However, the original design was to allow the synthesis engine to produce blocks that requested one or more scratch registers as parameters. In this environment it would be possible for the results to be incompatible with prior choices; for example, if we spilled two values before a given block to have two registers free, but the synthesizer decided it needs three scratch registers, we would want to go back and spill another. In the phase-based execution environment this is not possible. (Instead we have to provide explicit scratch register hints in synthesis blocks.)

Note that the implementation also relies on Grouper itself being deterministic, so that after restarting in each phase it can rerun analyses and transforms and get the same results, instead of needing to save its state.

7.6.2 Limitations of Grouper

One of the chief limitations of Grouper as it has been designed and built is that all the steps in a procedure must be machine-independent. That is, each block must exist (even if empty) for every machine. This limitation arises partly from the need to have a predictable collection of output block specifications (so that it is possible to drive a build) and partly from a lack of refinement machinery for working with the procedure body.

It is also difficult to write machine-dependent Grouper specifications, not for any fundamental reason but because none of the tooling was set up to allow it.

Grouper is really intended for, and only really works with, register machines; registers and random access to registers are built into the structure of its separation logic. One could build a similar tool for stack machines, but it would be much different. It also relies on there being enough registers to hold working values; in particular a procedure cannot have more arguments than there are registers to hold them. This is not a problem with modern architectures and even most old architectures, but is likely to cause problems working with 32-bit x86.

7.7 Future Work

Register allocation (picking otherwise unused registers to hold values that need to be saved) was part of the original design that has not yet been implemented. As discussed in Section 7.4.2, when a still-live value is overwritten, the value propagation pass saves this information with the intent of picking a register to put it in. Similarly, in cases where the "find" separation construct is unresolved, Grouper could pick a register for it instead of failing. Similarly, Grouper was also intended to be able to spill values to the stack if it runs out of registers. Because adding these extra conditions to arbitrary synthesis blocks might exceed scaling limits, the plan early on was to insert additional synthesis blocks whenever spilling or moves were needed, but this interacts badly with the phase model (and the need to have deterministic build rules) and a different approach is needed.

Similarly, there was a plan at one point to extend Capstan to be able to synthesize blocks and produce results in terms of some number of scratch registers it would determine. (It might do this by extending its CEGIS iteration to try additional numbers of scratch registers as well as additional numbers of instructions.) This would dovetail neatly with register allocation in Grouper: each synthesis block would ask for some number of scratch registers, and Grouper would either provide them directly or spill values to make room, all without any direct intervention needed either in the Grouper specification itself or in the machine-dependent material. This too interacts badly with the need to have deterministic build rules, however.

The long-term solution is to make Grouper run on-line, in the sense that it runs the other generation tools and the synthesis engine under the covers. Then the phases can be eliminated. It also becomes possible to insert additional synthesis blocks on the fly when needed, such as to spill values. Furthermore one can also try iterative approaches, which are currently ruled out by the phase model. One could also imagine an interactive version where the porter or the OS architect adjusts the specification on the fly to see how the output code is affeced.

In a different direction, generalizing the block model would be useful. Right now everything is done in very specific ways for the specific tools I have. I started out in that direction at one point but concluded it was premature then; now it would be somewhat less so. One way to begin is to add more tools. One could imagine various additional block generators; for example, a compiler for a simple procedural language intended to work in blocks (rather than whole procedures like normal languages) would be a useful way to generate code where machine-dependent considerations are scattered through otherwise machine-independent code. One could also imagine such a compiler either producing specifications (like Nautilus and Grayling) or accepting them (like synthesis blocks), which would require considerably different handling in Grouper. The real way forward in the long run is to figure out a model for interacting with block generators that these tools are specal cases of. However, this is going to be difficult. Right now the fact that Nautilus specifications are integrated before the locations files are generated for Grayling is vital to generating those files with minimal user effort; it is not clear how to abstract or generalize this interaction.

A third direction of future work is deploying Grouper into a verified kernel. In a verified kernel, the machine-independent kernel would provide a C-level specification for a whole Grouper procedure. This would take the form of a complete precondition and postcondition for the whole procedure. Nautilus (or actually a different tool with much more brain than Nautilus) would, as part of generating the code for a C linkage, also compile the machine-independent specification and merge it into the output `.grrbl` file. One would then write a Grouper specification as usual (with sparse assertions at individual program points) and by the soundness result the synthesis blocks would be forced to conform to the external specification or fail. This sounds like an at least moderately appealing way to approach producing verified assembly code for a verified kernel.

I would also like to experiment with Grouper's model of step-by-step specifications in the context purely of verification, both for assembly language and more generally. It would not be difficult to implement a verification mode for Grouper, where it reads a handwritten assembly procedure annotated with program points corresponding to a specification, and verifies the code against that specification. It might even be possible to infer the locations of the program points. More broadly, I have long felt that Hoare logic specifications for nontrivial procedures are at best unwieldy and the design of Grouper arose to some extent from frustration with them. Experimenting with verification of procedural languages using Grouper-like specifications seems to be a promising research direction.

Chapter 8

Deployment and Evaluation

As proof of concept and to test my claims, I have deployed the Aquarium toolset in an actual operating system in order to apply it in practice and find out how well it works to generate ports.

The deployment part involves both adapting Goldfish for synthesis and writing specifications for that synthesis. This is discussed in Sections 8.2 and 8.3. It also requires writing machine descriptions, discussed in Section 8.4. The evaluation consists of running the synthesis (and the compilation, and templating) and testing the resulting kernel. This is covered in Sections 8.5 and 8.6. The first section provides background on Goldfish itself.

8.1 Goldfish: A Small, Friendly OS

The operating system I have constructed to test Aquarium is called Goldfish. It is derived from OS/161, a teaching operating system [14] I first wrote in 2001. The current OS/161 is a multiprocessor, fully preemptible kernel supporting the basic Unix system calls. It comes with a simple Unix-like C library and a small set of user programs. It also includes a suite of tests and benchmarks accumulated over years of teaching. It has had substantial work put into it to make its internal interfaces and abstractions clean and well formed, including for the most part its machine-dependent interfaces. It is, however, intended for teaching and large sections of it are meant to be written by students. The basis for Goldfish includes the solution code for this material, which I have also written and maintained over the years and made comparatively clean and well formed. However, the *scope* of solution material is intentionally restricted: it is meant to be a shiny, polished version of what students can reasonably be expected to write. This affects the machine-dependent code and interfaces in three major places: first, the code for dispatching system calls, that is, the machine-dependent code for finding and extracting system call arguments and dispatching to machine-independent system call handlers; second, the virtual memory system, and finally, cache handling.

In most research kernels built since at least the early 1990s the code for unmarshalling complex system call arguments (e.g. copying string arguments such as pathnames into the kernel, and copying structure or array results out of the kernel) is automatically generated. The code for extracting arguments from the userspace stack or from registers saved in the trapframe is usually part of this; the code generator knows about the calling conventions for system calls (which are not always the same as the normal C calling conventions) and outputs the proper lookups. In OS/161, however, much like earlier kernels, the unmarshalling code is handwritten by students, as this has proven a valuable exercise for teaching them about address spaces, safety and security, kernel robustness, how pointers actually work, and various other character-building topics. The machine-dependent argument extraction code is also handwritten. The solution code for all of this, consequently, is also handwritten. This necessitated a porting solution for the machine-dependent argument extraction. This is not of particular research interest for this project (since as noted the standard approach is already code generation and it qualifies as a solved problem) so I set up an ad hoc scheme that uses C preprocessor macros paramaterized via Tuna files.

The portability of the virtual memory system *is* of research interest, and (unfortunately) the solution code for it was never intended to be portable. It is also lacking today-standard headline features: it does not support copy-on-write forks, memory-mapped files, or a unified buffer cache. Asking students to write a virtual memory system in two to three weeks is aggressive enough without introducing these additional complications, and the solution set reflects this. However, for this project, portability is not optional. So the first step distinguishing Goldfish from a copy of OS/161 with its solution sets was rearranging the virtual memory system to have a machine-independent part and a machine-dependent part. The results are described in more detail in following sections. The other virtual memory features are nice to have but not essential, so I have not taken the time to implement them. Most of them have little effect on the machine-dependent material in any event.

Cache handling is largely part of the virtual memory system but qualifies as a separate topic: OS/161 essentially does not support cache handling at all. Long ago we decided that manual control of (for example) VIPT caches was one step too much realism for students. Therefore there is effectively no cache handling in the solution virtual memory code or even in the OS/161 base system. Unlike the other virtual memory features, which are optimizations that one can live without, cache handling is *necessary* on many machines, and it was therefore necessary to add cache control logic to the Goldfish code. As of this writing, the process of doing so is only partially complete. The necessary cache operations exist, but the necessary calls to them at various points are not all in place. This will be at best an ongoing effort; missing cache flushes lead to intermittent bugs (sometimes very intermittent) and the only way to be confident that all have been found is extensive stress testing.

8.2 Specificity and Generality in Machine-Dependent Interfaces

Before going into specific details about the machine-dependent interfaces in Goldfish (and their relationship to interfaces in OS/161 and in BSD) I would like to point out a common theme.

Traditionally the machine-dependent interfaces in operating systems have been made as general as possible. This has partly served as design conservatism and future-proofing: the more general the interface is, the more likely it is to be able to work with succeeding generations of hardware. However, it has also often been expedience: often when we do not know what we need or know a way to express the range of options concisely, we express things as code. Designing general interfaces and hand-writing instances of them avoids the need to create explicit models and abstractions, at the expense of increased downstream costs for maintenance and for writing new instances. This is often a good initial tradeoff, especially since many things tend to remain unclear early in a design lifecycle. However, in such cases it should usually be revisited later.

As an example, early virtual memory systems were completely machine dependent. When Mach introduced the idea of a machine-independent virtual memory system [24], the reverse mapping structure (from physical pages to the virtual addresses using them) was left in the machine-dependent part ("pmap"). This might have been to leave open the possibility of architectures that needed different handling for this mapping, or because they wanted to concentrate on factoring out the structures they needed to advance their research goals, or because nobody had yet really articulated the idea that the reverse mapping structure is an inherent part of any virtual memory design, or any number of other reasons. (Note that Mach VM was published in 1987, which today is approximately half the lifespan of the industry ago, and well over half the time since virtual memory was invented, and experience has clarified a lot of things in the meantime.) But because the reverse mapping is always needed (to evict or even just unmap a page, it must be possible to find all the references to it) it must be reimplemented for every architecture.

This design property survived both NetBSD's and FreeBSD's separate virtual memory system redesigns in the 1990s. In both systems the reverse mapping is contained in the machine-dependent section of the physical page structure. This is `struct vm_page_md` in NetBSD, and `struct md_page` in FreeBSD, and consists of a list of per-mapping objects called, mostly, `struct pv_entry`[1]. In NetBSD the list is materialized in various ways on different machines, and sometimes the structure contains a reference to an address space as well as a virtual address (in other cases the address space is apparently not needed), and there are two outlier machines where the list is stored elsewhere (IBM 4xx-flavor PowerPCs and the VAX port), but in all cases it works the same way and could be replaced by machine-independent code. (NetBSD has in fact begun the process of doing so; machine-independent pmap elements may be found in `sys/uvm/pmap` in the NetBSD sources.) The various FreeBSD ports are much the same, except that fewer of them carry an address space reference and for SPARC the structure has a different name (`struct tte`).

One would expect that a new design today (or even some time ago) would include the reverse mapping in the machine-independent code. It is now clear that it is always needed, and it is also clear that we should not expect anyone to invent an exotic MMU where the machine-independent reverse mapping would be redundant with machine-specific structures. Consequently the interface in Goldfish places the reverse mapping in the machine-independent code.

This is not just a design refinement. Adding unnecessary requirements of this sort to the machine-dependent virtual memory code is a burden on human programmers, though not a huge one. (Although to some extent the cumulative effect of such underdesign is reflected in the fact that one traditionally writes a pmap module for a new machine by pasting a copy of an existing one and then editing it [18].)

For synthesis, however, unnecessary generality becomes extremely expensive. In the Mach (and NetBSD and FreeBSD) pmap interfaces, the need for the reverse mapping is implicit in the specification of some of the pmap operations. Successful synthesis requires, at a minimum, identifying this requirement and factoring it out into a standalone set of synthesis problems. Otherwise it is unlikely the pmap operations can be successfully synthesized. Synthesizing data structures internal

[1] Code references are to both projects' development heads as of February 2020.

to an interface might or might not be beyond the capability of some recent synthesis work, but it is definitely beyond our assembly language synthesis.

Thus for Goldfish, finding and removing unnecessary generality in the machine-dependent interfaces has been a significant fraction of the necessary work, part of (and difficult to disentangle from) the process of writing specifications for the interface.

8.3 Goldfish Machine-Dependent Interfaces

A full port of Goldfish currently requires fourteen header files, seventeen source files, and three miscellaneous files. These cover the following topics:

- Basic machine properties
- Build configuration (and compiler flags)
- Standard C functions
- Low-level CPU operations, including spinlocks, memory barriers, and interrupt control
- CPU identification
- Timers
- Kernel threads
- Traps
- User system call stubs
- System call ABI definitions
- User startup code
- Cache control
- Virtual memory
- Kernel startup code

The code sizes associated with these categories are shown in Figure 8.1. The table shows the number of files (and their types) that need to be generated for a port, the number of entry points into the machine-dependent code, and line counts for the handwritten MIPS port. The handwritten MIPS code is written to the same interfaces as the generated code. However, it does not, in all cases, always have the same internal structure; it represents the minimal changes from the original OS/161 MIPS code to match the specifications. In general it uses more header files (and more symbolic constants) than the generated MIPS port, and it omits entirely certain files that are empty in the generated MIPS

	Files:			Entry points to machine-dependent code:		Handwritten lines for MIPS:		
Category	`.c/.S`	`.h`	Others	C	Asm	C	Asm	Other
Basic machine properties	-	4	-	-	-	36	-	-
Build configuration	-	-	3	-	-	-	-	79
Standard C functions	1	1	-	-	2	5	41	-
Low-level CPU operations	1	2	-	2/0	6/8[1]	175[3]	-	-
CPU identification	1	-	-	-	1	62[3]	-	-
Timers	1	-	-	-	2	25[3]	-	-
Kernel threads	3	3	-	2	2	54	49	-
Traps	3	1	-	5	4	265	229	-
User system call stubs	1	-	-	-	2	-	28	-
System call ABI definitions	-	1	-	-	-	21	-	-
User startup code	1	-	-	-	1	-	23	-
Cache control	1	1	-	-	12	11	31[4]	-
Virtual memory	3	1	-	2	5/9[2]	163	101	-
Kernel startup code	1	-	-	-	2	-	96	-
Totals	17	15	3	11/9	39/45	817	598	79

Figure 8.1: *Sizes of Goldfish machine-dependent interfaces. Line counts were taken with* `wc -l` *after removing comments and blank lines, but not C preprocessor directives.*

[1] *Two operations can be implemented in C on most machines; see discussion.*

[2] *Four additional entry points are required on some machines; see discussion.*

[3] *Includes inline assembler.*

[4] *Stubbed out in the handwritten code; a real implementation would be larger.*

port. It also uses inline assembler for a number of procedures, which the generated code mostly does not. The line counts are intended to give a general indication of the size of the problem; close comparisons to generated line counts may be somewhat misleading.

The entry points are divided into those that can be implemented in C (with machine-dependent parameterization provided by combinations of templating with Hermitcrab and the C preprocessor) and those that require assembly. For some entry points this depends on the machine; conditional Hermitcrab logic driven by Tuna files is used to handle the various cases.

For completeness, note that there is also an additional 35-line header file (`regdefs.h`), effectively part of the MIPS assembly language, that is provided manually in both the handwritten and generated MIPS port; it is not counted in the table.

Figure 8.2 shows the breakdown of the assembler entry points by block type for the categories that involve at least one. Blocks may be canned (provided by the porter in a Tuna file), generated by Grayling (that is, compiled from a context specification), generated by Nautilus (that is, extracted as a call to a C function or a prologue/epilogue pair to be callable as a C function), or synthesized by

	Assembler	Code blocks:				
Category	entry points	Canned	Grayling	Nautilus	Synthesized	Total
Standard C functions	2	-	2	4	8	14
Low-level CPU operations	8	6	-	-	5	11
CPU identification	1	*see text*				
Timers[1]	2	-	-	-	2	2
Kernel threads	2	-	2	3	11	16
Traps[2]	4	1	2	1	10	14
User system call stubs	2	1	-	-	2	3
User startup code	1	-	-	2	8	10
Cache control[1]	12	-	-	24	12	12
Virtual memory[2]	9	-	-	14	11	25
Kernel startup code	2	*see text*				
Totals	41	8	6	48	69	107

Figure 8.2: *Code generation summary for Goldfish machine-dependent interfaces. Entry points are from Figure 8.1. These numbers are the same across all machines, though for most machines some blocks will be empty.*

[1] *These specifications are currently incomplete, but the number of blocks is unlikely to change.*

[2] *These specifications are currently incomplete; the number of synthesis blocks is likely to increase when completed.*

Capstan. Most of the entry points are Grouper procedures. Only two are not, the ones for system call stubs. For reasons discussed below they are generated from standalone Alewife specifications and composed with Hermitcrab.

Figure 8.3 shows the sizes of the specifications for these categories, broken down by tool. Notice that a large fraction of the specification material is output templates. Some of this reflects the size of the machine-invariant parts of output files (such as the build configuration templates) and some of it reflects the power and flexibility of templated C code.

The remainder of this section will go over these interfaces. For each category I will explain the purpose and contents, and discuss how they differ from the comparable interfaces in the original OS/161 and in NetBSD. (NetBSD is a comparatively well-structured example of a deployed production OS.) Then I will describe how the implementation is generated in Goldfish and discuss the specifications required. The specifications themselves may be found among the artefacts in the Goldfish tree.

8.3.1 Basic Machine Properties

This category covers the files `types.h`, `kerntypes.h`, `endian.h`, and `elf.h`. These are effectively the same as in OS/161, where the interface is fundamentally the same than the equivalent material in NetBSD or FreeBSD. For example, `kerntypes.h` is roughly comparable to `machine/ansi.h`

Category	Nautilus	Grayling	Grouper	Hermitcrab	Others	Total
Basic machine properties				60		60
Build configuration				177		177
Standard C functions	5	17	63	24		109
Low-level CPU operations			67	112		179
Timers*			28	9		37
Kernel threads	7	13	90	137	4[1]	251
Traps*	3	12	105	71		197
User system call stubs				37	30[2]	67
System call ABI definitions				68		68
User startup code	4		63	6		73
Cache control*	25		172	109	118[3]	424
Virtual memory*	10		32	321	2[4]	365
Totals	54	42	620	1131	160	2007

Figure 8.3: *Specification file line counts, taken with* `wc -l` *after removing comments and blank lines. These files are all machine-independent.*

[1] *A Tuna file defining the options for handling the current thread.*

[2] *Two single-block specifications written in Alewife.*

[3] *A one-line Tuna file defining cache types; the rest is the machine-independent cache model.*

[4] *A Tuna file defining virtual memory options.*

**These specifications are incomplete; the line counts are provisional and probably underestimates.*

in BSD, and `endian.h` is effectively identical to `endian_machdep.h` in NetBSD. (The two `types` headers define basic types such as `uint32_t`; `endian.h` defines the machine endianness; and `elf.h` defines the machine type code for the ELF executable format.)

In Goldfish the material is defined in Tuna files and the header files are filled in by Hermitcrab. For example, the template for `endian.h` reads a boolean that defines the endianness and outputs either `#define _BYTE_ORDER _LITTLE_ENDIAN` or `#define _BYTE_ORDER _BIG_ENDIAN`. This is neither simpler nor easier (nor more complex nor harder) than creating the header file by hand; either way the definition for a new port takes at most a few seconds to write. (I set this up with Tuna files for Goldfish mostly so that everything works the same way. Also because some of the information is also needed by other Aquarium tools and can then be read from the same Tuna files.)

8.3.2 Build Configuration

This category covers three pieces from OS/161: the machine-dependent part of the kernel configuration system, a file that hooks into the build system makefiles to allow setting compiler flags, and the linker script for linking the kernel. These files are all mostly the same from port to port and for Goldfish are templated using Hermitcrab with parameterizations from Tuna files.

Producing Goldfish led to discovering and fixing some weaknesses in OS/161's kernel configuration scheme; these changes will be merged upstream. In OS/161 the machine-dependent kernel configuration already had certain pieces that were meant to be included or not depending on each machine's needs; this mechanism has been systematized in Goldfish. Similarly there were some comments to this effect in the linker script; these are now templated hooks.

These interfaces are also much the same in NetBSD; OS/161 is BSD-like in its general layout, and that extends to both its build system and kernel configuration scheme.

The specifications are Hermitcrab templates. The only aspect of interest is that the use of templating allows certain definitions to be centralized. In particular, on some machines the best way to keep track of the current thread or CPU is to reserve a general-purpose register (more about that in Section 8.3.7) – the identity of the register needs to appear both in the code and as a compiler flag. In OS/161 these must be specified separately in different places and remain consistent; in Goldfish the choice of register appears in only one place (`kerndefs.tuna`).

The use of code generation enables this type of systematization, both here (where relatively simple definitions can be inserted by templating) and in more complicated cases. It is common in machine-dependent kernel code for single design choices to have ramifications in multiple places that must remain consistent; being able to specify the choice once offers advantages for maintenance.

8.3.3 Standard C Functions

There are two principal functions in Standard C that require implementations in assembler (because they manipulate the stack and save register values): `setjmp` and `longjmp`. The interface to these is standard and unchanged in Goldfish. They are also (in OS/161 and most systems) accompanied by a machine-dependent header file that defines the type they use; this is also unchanged in Goldfish.

Each of these functions is specified as a Grouper procedure. Each comprises a context block generated with Grayling and prologue/epilogue blocks from Nautilus, and four synthesized blocks, all of which are inserted by Grouper while integrating the external blocks. Three of the synthesis blocks need do nothing (at least on MIPS); the fourth sets the return value. For `longjmp` the return value is passed as an argument, except that it is forced to 1 if the caller passes 0. The specification for this runs afoul of the problem with value propagation and indirect dependences discussed at the bottom of Section 7.4.2. There are various possible workarounds; I chose to force the computation of the return value to an earlier block.

An additional template file pastes the Grouper-generated full procedures into an assembly source file and creates `setjmp.h` containing the `jmp_buf` context type used by the functions.

8.3.4 Low-level CPU Operations

There are four groups of procedures performing low-level CPU operations: memory barriers (`membar.h`), spinlocks (`spinlock.h`), interrupt control, and CPU idle/halt. I have retained the OS/161 interfaces in Goldfish.

OS/161 defines the following memory barriers as inline functions:

- `membar_load_load`
- `membar_store_store`
- `membar_store_any`
- `membar_any_store`

Each of these separates a set of operations issued before the barrier from a set of operations after the barrier; for example, `membar_store_any` ensures that stores issued before the barrier take effect before either loads or stores after the barrier. The load/load and store/store barriers are used for ordering reads (or writes, respectively) to device registers. The store/any and any/store barriers are used when acquiring and releasing locks, respectively. The other possible combinations are not used by the machine-independent code and thus do not need to exist. This interface, alas, fails to distinguish ordering barriers from flushing barriers, that is, barriers that just cause memory accesses to be ordered from barriers that guarantee that accesses have been presented to the memory system before anything else happens. The former are fine for locking, but accesses to device registers require the latter. (This is a rare example of the machine-dependent abstractions in OS/161 being inadequate rather than over-general. It will be fixed upstream.)

I have chosen to define an additional barrier `membar_any_any` and implement the others using it. This is for three reasons: (1) the issues raised in the previous paragraph, (2) because using a stronger barrier is always safe (just slower), and (3) because the goal of this project is a proof of concept. (The handwritten MIPS code from OS/161 also does this, because the version of the MIPS architecture it supports has only one memory barrier instruction, with that semantics.) I do not attempt to *synthesize* memory barriers, as explained in Section 4.3. The function is generated with a template for a C header file using inline assembler, with the instruction itself provided as a canned block in a Tuna file. (This is an expedient way to produce a simple function when no synthesis is needed.)

This memory barrier model is based on the SPARC memory barrier instruction, which is clear and easily understandable. NetBSD uses a slightly larger set of barriers with different and more confusing names; for example, its `membar_consumer` is OS/161's `membar_load_load`. The C11 standard has another programming model entirely. Memory barrier semantics and programming models are a complex topic beyond the scope of this project.

OS/161 has the following machine-dependent spinlock interface:

- `spinlock_data_set`
- `spinlock_data_get`
- `spinlock_data_testandset`

These assign to, read from, and atomically test-and-set a value of type `spinlock_data_t` and are defined as inline functions. They are used in a more complex machine-independent spinlock implementation that also handles interrupt levels, memory barriers, and has an optional deadlock detector.

(The test-and-set operation must be atomic, but is assumed not to contain any memory barriers.) This is perhaps not the best interface; in the past, tinkering with it has produced noticeable speedups. However, it serves for OS/161 and will serve for Goldfish. On most machines (with non-hostile compilers) the first two functions can be implemented in C and only the atomic test-and-set requires assembly. I have kept this format for Goldfish. Note that test-and-set (rather than compare-and-swap) is more portable. All current architectures, and with high probability all future architectures, support compare-and-swap, but some old architectures only support test-and-set. (On old architectures with no multiprocessor support at all, the interrupt handling in the machine-independent code means the test-and-set operation need not be atomic.) I do not synthesize the test-and-set operation; see Section 4.3. The spinlock interface is generated as templated inline assembler in C header files, using canned definitions from Tuna files. The Tuna definitions for the `spinlock_data_set` and `spinlock_data_get` have type `option multiline`; if this is set to `None` the C version is used.

Interrupt control in OS/161 and Goldfish is accomplished with these machine-dependent functions:

- cpu_irqon
- cpu_irqoff

These are the back-end implementation of a machine-independent BSD-style spl interface, except that unlike BSD kernels OS/161 does not have interrupt priorities. That is, interrupts can be either on (spl0) or off (splhigh). I chose this design for OS/161 in 2001 to reduce unnecessary machine-dependent code and decided that interrupt levels were unnecessary for teaching and possibly not even useful for production [28]. In NetBSD the full spl interface is reimplemented separately for each machine.

The specifications for cpu_irqon and cpu_irqoff are Grouper procedures using state abstraction: an abstracted 1-bit register representing the interrupt state is imported from a Tuna file. On MIPS it can be instantiated directly with a 1-bit register (which is a field of the status control register), but on RISC-V and ARM it needs to combine two bits. (There are separate interrupt control bits for user mode and kernel mode.) The specification can then set this register to either 0 or 1 in the postcondition of a single synthesis block.

The remaining entry points in this category are cpu_idle and cpu_halt. The first is for idling the current processor when there is no work to do. The second diverges; it is used as part of system shutdown. These are both specified in terms of a single canned code block that executes the machine's idle instruction, which waits until an interrupt arrives. (If there is no such instruction, it can execute a nop instead.)

For cpu_idle the Grouper specification also imports two booleans, called irqon_during_idle and irqon_after_idle. The first inserts logic that enables interrupts while executing the idle instruction. The second inserts logic that enables interrupts briefly *after* executing the idle instruction. Either of these might be needed on some machines for the incoming interrupt to be delivered and serviced. (The interrupt must be handled in order that some thread be awakened; otherwise the idle loop never ends.) The Grouper specification for cpu_idle was shown in Figures 7.1 and 7.2 in the previous chapter.

For cpu_halt, there are no interrupt-related considerations to worry about, but since the function is supposed to diverge, it requires a loop to wrap around the idle instruction. Since control flow support in Grouper is not implemented yet, the loop is implemented as a literal block containing an assembler label and a canned block with an unconditional branch instruction.

There is also a Hermitcrab template that combines these last four procedures into a single assembly source file. Note for the record: in Goldfish, these functions appear in a file `cpuasm.S`. In OS/161 they use more inline assembly and are combined (along with CPU identification and some trap-related things) in a single file `cpu.c`.

8.3.5 CPU Identification

The OS/161 CPU identification interface is a single function that examines the CPU for interesting properties and prints them into a string buffer. However, it does not, for example, check the CPU model to choose different runtime behavior for different machine architecture variants. OS/161 has no support or infrastructure for that or for dealing with architecture variants at all. I have not added any such support to Goldfish, because none of the project infrastructure has any support or models for reasoning about machine architecture variants either; we have been treating variants as different machines and postponing a more detailed treatment to future work. It is not yet clear to what extent such support is even necessary in a world where ports can be synthesized. On the one hand, it is no longer important to share code between kernels for different variants when the code is no longer hand-maintained. On the other hand, it remains desirable to have one binary kernel image that boots and works on as many physical machines as possible. Needing a different kernel image for each minor x86 variant that has appeared in recent years would be a configuration management disaster from an end-user standpoint. (And for virtual machines running on cloud providers, perhaps worse, as the specific machine variant running your virtual machine can change with little notice.)

It is also not clear what a good systematic interface for this functionality is, if any. In many cases, architecture variants that require different behavior must be probed early in kernel startup, so that subsequent logic in the kernel startup code can take the proper paths.

In NetBSD there is no systematic scheme for this, or at least none that is machine-independent; e.g., the MIPS port has a complex system of preprocessor logic for transparently testing variant architecture properties at either compile time or runtime, and the x86 port has support for hotpatching various low-level kernel functions based on the architecture version, but this code is all machine-specific. The user-level program that prints CPU models and features for user consumption has an interface much like the OS/161 one, with ad hoc code for each supported architecture.

Meanwhile, specifying the code to run is itself difficult. The set of machine architecture versions to probe for is machine-specific, as are the register or instruction properties to inspect for each probe and the proper sequence for performing the probes to avoid getting wrong answers. On some machines these probes are themselves difficult to synthesize or reason about; for example, the x86 `cpuid` instruction writes strings into memory. As of this writing, we are not certain if Cassiopea can represent this instruction fully, or if we can represent a specification that will cause the synthesizer to choose it, or if having done so it is possible to write a specification that does something useful with the result without exceeding the current scaling limits and without requiring properties intentionally left out of the synthesis engine such as the ability to generate loops.

For these reasons I have postponed CPU identification to future work. I have stubbed out the `cpu_identify` function in Goldfish and not attempted to generate it.

8.3.6 Timers

Time handling is another area where OS/161 is weak. It assumes that each CPU has an on-chip (or at least, per-CPU or per-core) timer that it can program with a fixed value to provide time-slicing interrupts. It has no support for measuring the tick rate of this timer or calibrating it in any way. It also has no interface for it; the timer is initialized as part of the mainbus initialization code, and rearmed after interrupting as part of the interrupt dispatching logic, using inline assembler. This was not entirely suitable for Goldfish purposes; the bus handling code is out of scope, but by being so it is not supposed to have CPU control hooks in it. Therefore I refactored it and produced the following simple interface:

- `machdep_hardclock_bootstrap`
- `machdep_hardclock_rearm`

which take no arguments and return nothing. This is not really an adequate interface for proper time support, but as OS/161 and thus Goldfish does not really have proper time support anyhow, it will serve for the time being.

Note that NetBSD's timing interface is complicated by comparison; CPUs may have multiple "timecounters" with varying precisions, which are chosen at runtime based on fairly complex criteria.

Because of concerns that the specification might require adding support for temporal logic or other expensive features to the implementation, work on this category was deferred. This now seems unlikely to be a problem, especially since the tick value to program the timer with can be a constant. I think at worst I might need to develop a lightweight model for programming and arming the timer. Nonetheless for the time being the Grouper specifications are left to future work.

As with many of the other categories, there is an additional trivial Hermitcrab specification to create a single output assembly source file.

8.3.7 Kernel Threads

Unlike time handling, the kernel thread system in OS/161 is mature and reasonably complete. There are six machine-dependent files:

- `current.h`: defines `curthread` and `curcpu`
- `switchframe.h`: defines the context structure for thread switch
- `switch.S`: the thread switch code
- `threadstart.S`: code that runs when a new thread first starts
- `thread.h`: defines the machine-dependent part of the thread structure
- `thread_machdep.c`: code for handling the machine-dependent part of the thread structure

Section 7.3.1 already explained the issues associated with tracking the current thread or CPU. In the original OS/161 the machine-dependent code can either provide a definition for `curthread` (the current thread) or a definition for `curcpu` (the current CPU or core), and the machine-independent thread code adjusts as needed. Goldfish extends this to explicitly support the three most likely approaches: The three methods are:

- Reserve a register (general-purpose or otherwise) to hold `curthread`.
- Reserve a register (general-purpose or otherwise) to hold `curcpu`.
- Make `curcpu` a constant address by creating a per-CPU virtual memory mapping at a well-known address. (The virtual memory system's portion of the support for this option is, unfortunately, incomplete at this time.)

The specification refinement method described in Section 7.3.1, but in this case running in Hermitcrab rather than Grouper, allows writing a single Hermitcrab template to handle all three cases. The options are defined as constructors of a sum type in a machine-independent Tuna file. For type safety, constructor arguments give the register or address needed. A machine-dependent Tuna file imports that and applies one of the constructors. This file is imported by Hermitcrab, which matches on the constructed value to distinguish the cases. Thus, in each case a `current.h` is emitted with the right material in it. For the cases that reserve a register, the proper compiler flag is also added to the kernel build as mentioned in Section 8.3.2.

The thread switch interface in OS/161 is a structure of type `struct switchframe` and an assembly procedure `switchframe_switch` that accepts two slots that hold switchframe pointers (in the thread structure) and does the machine-level thread switch, allocating the context structure on the stack. This interface is exactly the same in Goldfish. The comparable NetBSD interface is `cpu_switchto` and differs only in details – the context structure is an array rather than a struct and is an explicit member of the thread ("lwp", lightweight process) structure rather than being allocated transiently on the stack. The specification is a Grouper procedure using two Grayling blocks (first to save the previous thread's registers, then to load the next) as well as several synthesis blocks and a C linkage created by Nautilus. This procedure requires the Grouper hook described in Section 7.4.1 for disabling the usual assertions about preserving callee-save registers. Thread switch does not preserve callee-save registers, in the sense that on each call the callee-save register values on return are from a different thread and thus (in general) different from those on entry, even though from each *thread*'s point of view the values are preserved. The specification must enforce this some other way; fortunately the entry and exit assertions provided by Grayling serve this purpose so nothing needs to be written out in longhand. (See Section 6.7.2 as well as Section 7.4.1.)

The thread start code in OS/161 is an assembly procedure `switchframe_threadstart` that serves as a trampoline for calling some other function in a new thread. It expects to find the function to call and two arguments (one integer and one pointer, to discourage students from casting between integers and pointers) in registers; it is responsible for finding these, moving them to the proper places for arguments, initializing the stack as needed, and calling the requested function. There is a corresponding C function `switchframe_init` that initializes a switchframe structure such that when the thread switch code loads it, execution begins at `switchframe_threadstart` with the proper values in the proper registers.

The Goldfish interface is the same, except that I made `switchframe_init` machine-independent via the magic of templating and the C preprocessor: the fields for the proper registers are now accessed through macros defined in `switchframe.h`, and all the rest are zeroed. (The registers in question are specified in a Tuna file; they must be callee-save registers because the thread switch code only saves callee-save registers.) The Grouper specification for `switchframe_threadstart` has an assembly linkage (because its arguments arrive in these callee-save registers), one synthesis block (whose job is to clear the return address and if necessary adjust the stack pointer), and a Nautilus block for passing the arguments to the machine-independent `thread_startup` entry point.

The NetBSD interface corresponding to switchframe_init is cpu_lwp_fork; it works somewhat differently and includes some trap handling and userspace-related material (for the fork system call) that is kept separate in OS/161 and Goldfish. lwp_trampoline is, mostly, the equivalent of switchframe_threadstart, though not all ports call it that and at least one has multiple versions for different machine architecture variants. This arises because it is agglomerated with the code for returning to user level in a child process after fork.

Finally, in OS/161, most of the things traditionally put in the machine-dependent part of the thread structure are machine-independent (such as the pointer to the thread switch context) and for all ports so far there are only two elements optionally needed. These are related to user/kernel memory copying and are discussed in Section 8.3.13 below. If needed Hermitcrab places the declarations in thread.h and inserts initialization code into thread_machdep.c.

8.3.8 Traps

Trap handling can be divided into two parts: the first is the assembly-level exception handler that saves the machine state, and the second, typically written in machine-dependent C, is the logic for dispatching traps to the proper code for dealing with them. This separation exists in OS/161 as well as Goldfish (and also in NetBSD). The exception handler material can also be divided into two portions: the handler code itself, which is relatively clear-cut, and a more nebulous surrounding environment that connects the hardware to the handler code. The number of exception entry points varies with the architecture; so does the way these entry points are specified and how one arranges for them to execute the proper code.

For example, on MIPS, when most exceptions occur, execution continues at a single hardwired address. The kernel must accomodate this, typically by copying exception handler code into place early in kernel initialization. There is also a second exception entry point specifically for fast-path software TLB refill. This has a completely different set of requirements and is discussed later in this section.

The situation on RISC-V is simpler: there is, by default, one exception handler address for all exceptions, which is configurable via a control register.

x86 (perhaps unsurprisingly) is much more complicated. Each trap type has its own entry point. The processor's behavior is not uniform across these: when a trap occurs it pushes things on the kernel stack, but exactly what depends on the trap type. Meanwhile the addresses of these entry points are configured through an elaborate in-memory system of segment tables.

On machines with multiple trap entry points many kernels also use multiple copies of the context handling code, sometimes not all quite the same. For Goldfish this introduces too many complications. At the cost of a slight slowdown, Goldfish always uses exactly one copy of the main assembly trap handler with the context handling code. On machines with fixed-address entry points, it places a jump to the real handler at the fixed address. On machines with multiple entry points, each will explicitly push or load a code to report what happened and then jump to the real handler. When the multiple entry points are not uniform, it will attempt to compensate for the differences. (This can be done easily enough for x86, though I have not done an x86 implementation.)

It is still necessary to generate these pushes and jumps, which raises a problem: the number and nature of these is machine-dependent. It is not yet clear how to specify them in an effective way. For the time being I have deferred this problem to future work.

This raises another closely related issue, however. Trap handling, or some trap handling, has an important characteristic that it shares with the kernel startup code: parts of it can be machine-*specific*, not just machine-dependent. This arises (in both cases) because, to some extent, the code interfaces with the hardware rather than with the machine-independent kernel. This allows hardware properties for which the kernel has no machine-independent model (or where no machine-independent model exists) to appear in the interface. One example is that on SPARC the hardware generates a trap when it runs out of available register windows. Writing out ("spilling") one or more modified register windows and marking them available for reuse is an entirely machine-specific action taken in response to this machine-specific request. Register windows are not part of the kernel's machine-independent abstractions of machine-dependent phenomena, and the register window operations are not part of the kernel's machine-dependent interface. (In fact, both the machine-independent kernel and most of the machine-dependent kernel know nothing about them.)

I have not attempted to handle synthesis for machine-specific operations. It could be done by writing machine-specific specifications; from a pure code standpoint the machine operations for these are not markedly different from others. However, the infrastructure in Goldfish is not currently equipped to handle it, so it remains future work.

The main assembly-level trap handler, however, does the same thing on every machine: save the previous machine state into a trap frame and adjust the processor state for in-kernel execution, then call into the trap dispatching code. (And on return, restore the machine state and do a return-from-exception operation.) This is the same structure in Goldfish as in OS/161. (And NetBSD, and, in general, every kernel.)

The specification for the assembly trap handler is a Grouper procedure; it uses Grayling context blocks to save and restore registers, a Nautilus block to call the C-level trap dispatcher, and several synthesis blocks for other tasks, such as checking if the trap occurred in the kernel or not, locating the kernel stack, and allocating the trapframe on the kernel stack. Some of these blocks are empty on some machines; for example, on x86 the hardware takes care of finding the kernel stack, whereas on MIPS that must be done explicitly and is a nuisance. (OS/161, and hence Goldfish, has a feature to help with this: global arrays indexed by the CPU number that hold the kernel stack and current thread pointer while executing in user mode. In Goldfish these are enabled with a flag in one of the Tuna files.) Conversely, on x86, the context block must know where the hardware put the previous stack pointer value, because the hardware finds and loads the kernel stack. Somewhat similarly, on RISC-V the standard approach is to save the kernel stack pointer in a control register allocated for the purpose, and then exchange it with the original stack pointer. The Grayling partial saves feature (Section 6.8) is designed to handle these situations. As of this writing the Grouper specification for trap handling is incomplete.

Goldfish and OS/161 have an additional related entry point that jumps directly into the trap return sequence, which is used for the first transition to user mode in new processes after `fork` and after loading a new program in `exec`. As noted above in Section 8.3.7 in NetBSD this logic is combined with the kernel thread startup code rather than being factored out. The specification for this entry point has not actually been written yet.

The trap dispatching code in Goldfish is, for now, handwritten. Generating it (whether by synthesis or templating) requires a way to declare how to extract trap codes from the saved registers in the trapframe, and then a way to declare a mapping from trap codes to kernel actions to take. Currently Tuna files support finite maps, but not finite maps whose keys are machine integers. This will require future work.

Finally, a number of architectures (including MIPS) have a special trap entry point for fast-path handling of TLB refill exceptions. This trap is unlike standard traps: the goal is specifically not to save and restore registers and call into C code, but to handle the trap directly in assembler. To this end the handling code needs to walk the page table. This in turn requires the ability to branch to the slow path handler upon encountering a nonexistent page or subtree of the table. Currently the Aquarium toolset cannot handle this goal: control flow involving if-else statements is not yet implemented in Grouper. Meanwhile the Capstan synthesis engine cannot synthesize the conditional branches needed to implement if-else statements. Consequently the TLB refill handler is left to future work.

As with many other categories, there is a Hermitcrab specification that combines the various outputs into source and header files. The header file (`trapframe.h`) contains the trapframe declaration generated by Grayling. The assembly-level trap code goes in `exception.S`, and the C-level trap dispatching code goes in `trap.c`. The global arrays mentioned, if enabled, go into a file for machine-dependent CPU code, `cpu.c`.

8.3.9 User System Call Stubs

In Unix-type operating systems, system calls are presented to user-level code as C function calls. This requires that a small piece of code, known as a *stub*, be made available, usually as part of the C standard library. The stub is responsible for taking any steps needed to interface the kernel system call ABI to the C function call ABI, both on call and return, and also to trigger the system call itself.

The stub naturally divides into three blocks. Before the call, the stub normally must load the system call number into a register. After that it triggers the call itself with a special instruction. Finally, after the call it must interface the kernel's failure reporting mechanism with the user space one. Traditional Unix system calls return a value (in a register, or two registers for a double-width value) and a success or failure indication in the processor's carry flag. On success, the value is the system call's return value; on failure, the value is the error code describing the failure. The C function call, however, is expected to return the return value on success, and on error return -1 and leave the error code in the global variable `errno`.

Generating system call stubs raises some extra challenges. There must be one system call stub per system call, and there are dozens of system calls. Synthesizing each one separately, perhaps by cut-pasting a specification repeatedly, would dominate both the line counts and (more importantly) the synthesis time. Furthermore, most of the logic is the same in each stub; only the initial block that handles the system call number is different. OS/161 already had a scheme that shared the logic and produced the initial blocks from a template using the C preprocessor. I kept this scheme and adapted it for the Goldfish environment. This precluded using Grouper; instead the macro for the initial block is produced using a handwritten Alewife specification, and the error-handling block is a second handwritten Alewife specification. The system call instruction goes in a canned block. These are combined into a C preprocessor macro with Hermitcrab.

On some machines, however, the call number goes in the instruction, not in a register. This will not work with the current specifications.

For some OSes, on some machines, making system calls also involves loading more or different arguments into registers than the normal C calling conventions dictate. I have intentionally avoided this complication in Goldfish.

Generating the block that loads the system call number also poses a problem: the block must be parameterized by the system call number. Our synthesis engine cannot produce such blocks; it only allows concrete constant values. To solve this problem I synthesize one block with a single system call number, using a magic number that is both large enough to not appear by accident and small enough to fit into the immediate value fields of most processors. (The smallest useful such field I am aware of is ten bits wide; the largest system call number used in OS/161 and thus Goldfish is 119.) Then this number can be replaced with the desired number, or in fact the C preprocessor macro argument, by text substitution. This must match against decimal, hex, and binary forms of the number, as any might potentially appear. This substitution is currently done by a short `sed` invocation interposed between the synthesis run and the Hermitcrab run that creates the final output. (Ideally it would be done by Hermitcrab itself; more ideally, the synthesis engine would support parameterized blocks and it would not be necessary.)

This interface is shared by Goldfish, OS/161, NetBSD, historical Unix, and many other systems.

8.3.10 System Call ABI Definitions

The file `scabi.h` contains preprocessor definitions issued by Hermitcrab that parameterize the system call argument extraction code. As discussed in Section 8.1, this is an ad hoc solution for an OS/161-specific difficulty handling an issue that is best addressed a completely different way and need not be discussed further.

8.3.11 User Startup Code

The *startup code* for a C program is the code that runs before `main`. Traditionally called `crt0` (for ”C runtime”), the file `crt0.S` in Goldfish and OS/161, it has four major jobs: collect the arguments to `main`, perform any needed initialization of the C library, call `main`, and then call the standard C function `exit` with the return value of `main`, if any. On some OSes it is also involved in loading shared libraries, though this is now more commonly a separate additional layer.

The Goldfish and OS/161 user startup code is effectively identical. The interface differs from NetBSD only in details. Note that OS/161 (and hence Goldfish) does not support shared libraries or dynamic linking, so there is no equivalent of that separate additional layer.

This is mostly written in Grouper, with two Nautilus call blocks for calling `main` and `exit`. The specification imports a machine-dependent register map from a Tuna file in order to allow a machine-dependent set of register initializations. This handles initializing the global pointer register on machines that have one. In OS/161 the call to `exit` is wrapped in an infinite loop due to concerns about students' implementations of process exit not actually exiting and causing bizarre behavior upon return. I dropped this from Goldfish because it is not needed: Goldfish's `exit` works.

8.3.12 Cache Control

On some machines, the RAM cache requires some level of explicit handling. (See Chapter 2.) I have added cache handling and a machine-independent cache interface to Goldfish for the virtual memory system to use. It consists of these operations:

- `cache_vi_wb_page`
- `cache_vi_wbinv_page`
- `cache_vi_inv_page`
- `cache_vivt_flush`
- `cache_dma_wb`
- `cache_dma_inv`
- `cache_splitid_inv`

The first three operations are for virtually indexed caches (whether virtually or physically tagged): they flush a page by its virtual address and are meant for use when the virtual memory system manipulates pages. (The three forms write back, write back and invalidate, and just invalidate the cache mappings, respectively.) On PIPT caches these do nothing.

The `cache_vivt_flush` operation is for VIVT caches only and flushes the whole cache, or at least the whole VIVT part of the cache. This is done on MMU context switch so that one process's cache entries do not leak into the next process. On non-VIVT caches this operation does nothing.

The `dma` operations are for use before starting DMA writes and reads (respectively). Except for machines where DMA participates in the cache coherence scheme (which is mostly not the case), these flush the cache all the way down to main memory, so that DMA write operations fetch the intended data, and subsequent accesses to data placed by DMA read operations read that data and not stale values. They operate on a region of memory by virtual address and size, not necessarily a single page. Note that the devices in System/161 (OS/161's native platform) do not use DMA. (This was another intentional simplification for pedagogical purposes.) I added the DMA operations in anticipation of future ports to other hardware devices.

The `splitid` operation is for machines with split instruction and data caches. It invalidates the instruction cache for a range of memory by virtual address and size. On other machines it does nothing.

These operations are implemented as machine-independent C code (parameterized by Tuna definitions of cache types and sizes) in terms of the following operations synthesized in assembler:

- `icache_inv_vaddr`
- `icache_inv_index`
- `dcache_wb_vaddr`
- `dcache_wbinv_vaddr`
- `dcache_inv_vaddr`
- `dcache_inv_index`
- `l2cache_wb_vaddr`
- `l2cache_wbinv_vaddr`
- `l2cache_inv_vaddr`
- `l3cache_wb_vaddr`
- `l3cache_wbinv_vaddr`
- `l3cache_inv_vaddr`

As before, the operations may write back, write back and invalidate, or just invalidate. Each of these operates on a single cache line of the cache named, and addresses the cache either by virtual address or by index number (and way number). It is assumed that the L2 and L3 caches are PIPT and that the instruction cache is read-only. (This governs the set of operations needed.)

Note that this interface is a starting point; it is designed for our synthesis engine, which cannot generate loops. (Thus, the looping over addresses or cache indexes and ways must be done in C code.) It uses static definitions of cache type and geometry, which is insufficient in the real world: cache geometry typically varies not just by machine or machine variant, but from processor model to processor model of the same machine and machine variant version. Typically this cache information is intended to be probed at runtime. Note that while changing the C code to be parameterized at runtime rather than compile time (and directing this with the Tuna definitions) would be a straightforward matter of programming, generating the probe code is more challenging. Also there exist ARM SoCs whose cache geometry is not the same across all their cores[2], which would require additional infrastructure to handle.

Furthermore, looping in C code creates inefficiency on some (mostly older) machines. Cache flushing on vintage MIPS (MIPS-I) requires setting processor mode bits that are incompatible with most ordinary instruction execution and thus compiled code; flushing one cache line at a time requires toggling these mode bits for every cache line.

However, improving this lower-level interface when future work makes the synthesis engine more capable is relatively straightforward: it is not exposed to the rest of the operating system, only to the higher-level interface. I believe the higher-level interface to be suitable in the long term.

As already stated there is no comparable interface in OS/161. NetBSD does not have an interface like this; for the most part cache operations are defined on a machine-specific basis and referenced where needed in the per-machine pmap code. The new machine-independent pmap material mentioned in Section 8.2 does make use of per-machine icache flush operations. The DMA operations above are related to (but a subset of) NetBSD's `bus_dma_sync`, which also handles memory barriers, bounce buffering, and other bus-level considerations.

Writing specifications for cache control is problematic: one must provide the synthesis engine with an operational model of the cache that it can manipulate with the machine's cache control instructions. But a complete model of the inner workings of the cache, that keeps track of whether each cache line has valid data in it and so forth, is both impractical (the details vary from model to model, and are often not even documented) and infeasible (it would overwhelm the synthesis engine).

The critical insight is that the cache model given to the synthesis engine need only be detailed enough to allow it to choose the right cache operation out of the small number available. The

[2] For example, the Rockchip RK3399 (`http://rockchip.wikidot.com/rk3399`). Thanks to Jared McNeill of NetBSD for bringing this to my attention.

operations found in the Goldfish low-level interface, as just defined, are the only ones required during ordinary operation. (Others might be needed to initialize the cache at system startup, but that falls under the kernel startup code category and is left for future work.) There are three operations: write-back, write-back-and-invalidate, or just invalidate. These are done mostly by virtual address, but in some cases by index and way number. Therefore, the synthesis engine must be able to pick the desired operation, and it must be able to distinguish between the two addressing types, but none of the other details matter. In particular there is no need to care about *what* is in the cache.

I developed the following lightweight cache model that is small enough to permit efficient synthesis. (In fact, synthesis of cache control operations typically completes in seconds.)

- Each cache (L1 instruction cache, L1 data cache, L2 cache, and L3 cache) is modeled independently.
- Each cache has only two entries (that is, two individual cache lines) in it.
- I call these "blue" and "green" to distinguish them.
- Each of these entries can be in one of four states: clean data present, dirty data present, no (or invalid) data present but valid data exists in the layers below, or no data present and no valid data exists in the layers below either.
- A function is provided that chooses either the blue or green entry based on a virtual address.
- A function is provided that chooses either the blue or green entry based on a cache index number and way number.

The four entry states are required to distinguish the three cache operations; in particular the state with invalid underlying data is required to distinguish the "write back and invalidate" operation from the "just invalidate" operation. Note that these states do not correspond to the the states in the "MESI" (or any similar) cache model – those models serve an entirely different purpose (reasoning about consistency in a multiprocessor environment).

The functions can be any function from bitvectors of the appropriate sizes to boolean (I use "equal to zero") but they must be *different* functions. The Cassiopea machine description uses the cache model by defining instructions[3] that take a cache reference as an operand. The cache reference

[3] Some machines use instructions for cache control. Others use side effects of control register accesses, or even of memory accesses. Cassiopea can handle any of these; to avoid confusion the model description discusses only instructions.

can be either an address or an (index, way number) pair. The instruction definition uses one or the other of the model functions to pick a cache entry. (Then it updates the state of that cache entry as appropriate for the instruction, or for the operand that chooses the state transition.) Specifications, meanwhile, name a cache entry, by applying one of the functions from the model to procedure arguments, and name the desired state transition. The synthesizer must pick an instruction that makes the correct state transition. However, we also need it to pick an instruction that addresses the cache in the correct manner. It can only do this if it can distinguish the two addressing functions provided in the model. That is, if the two functions always map the procedure argument value to the same cache entry, the synthesizer cannot tell which to use and will pick an instruction using either one arbitrarily, and thus produce wrong code.

It is sufficient for the two functions to differ on only one input value, because the the input value is (implicitly) universally quantified in the specification. However, it is difficult to write machine-independent functions that will guarantee this under all circumstances. The bit sizes of the values involved are not known; they will differ from machine to machine. Furthermore, some instructions may not use all of the addressing bits. (For example, a cache control instruction that takes a virtual address in a register might split off the portion of the value that corresponds to the offset into a cache line and use it to hold the operation code.)

Consequently I have developed the following technique, which is of general use for building lightweight models. (It seems unlikely that it is novel, but I have not found it described elsewhere, likely because it is too obvious to get much attention.)

Let a *golem*[4] *variable* be an extra, artificial piece of machine state; essentially, an additional register that does not exist in the actual machine or the actual machine specification. For each function that needs to be distinguishable, create a one-bit golem variable. Incorporate its value in the output of the function such that changing the value of the golem variable always changes the output of the function, such as via exclusive-or. The specification will be tested for all combinations of values for the golem variables, and any two functions distinguished by golem variables will be different for some such combination, because in the synthesis the input machine state is universally quantified.

The functions used in the cache library are:

$$\texttt{if } \mathit{vaddr} = 0 \oplus \mathit{registers}[\texttt{vaddr\_golem}] \texttt{ then blue\_entry else green\_entry}$$

[4] The name originally arose as an in-joke relative to Skolem variables.

for access by virtual address and

$$\texttt{if } (\mathit{index} = 0 \wedge \mathit{way} = 0) \oplus \mathit{registers}[\texttt{vaddr\_golem}] \texttt{ then blue\_entry else green\_entry}$$

for access by index and way number.

These always behave differently for some starting machine state, no matter what the argument values provided are, so when a specification uses one or the other the synthesis engine will reliably always pick a cache control instruction that uses the same one. Thus, it can generate code that addresses the cache in the intended manner without needing a detailed functional model of the cache internals.

As of this writing I have successfully synthesized cache operations using Alewife specifications and Capstan, and also with Penguin, but the Grouper version of the cache model is unfinished.

8.3.13 Virtual Memory

As described in Section 8.1, OS/161 does not natively have a machine-independent virtual memory system. Its virtual memory system is MIPS-specific and relies on MIPS having a software-refilled TLB. It has a forward mapping structure of objects and pages belonging to objects and a reverse mapping structure, but where one would ordinarily also have page tables, it just loads translations from the forward mapping structure directly into the TLB. (This is a simplification we have always encouraged our students to make as well.)

Consequently to set up a machine-independent virtual memory system for Goldfish I needed to first change this to use page tables, then divide it into machine-dependent and machine-independent parts. I followed the cue of Linux, where the page tables are a mostly machine-independent structure. (Linux internals change regularly and without notice, but this architecture still remains as of Linux 5.5.6.) This is a plausible approach, because on essentially every machine, page tables are a fixed-depth radix tree, with between two and five layers. (32-bit machines typically use two layers; x86_64 is now up to five.) Using a combination of Hermitcrab templates and C preprocessor definitions I wrote a machine-independent page table implementation parameterized by various properties of the machine: the number of layers, masks and bit counts corresponding to the layers, extraction of the pointer part of each internal layer, whether that pointer is a physical or virtual address, etc.

I also made the reverse mapping structure machine independent (as discussed in Section 8.2) as well as most of the early bootup code that figures out how much RAM is available.

This leaves three groups of functions requiring machine-dependent implementations: the `copyin`/`copyout` family (for transferring user data into and out of the kernel), a group of functions for handling the TLB, and another group of functions for managing temporary mappings of physical pages for kernel access.

`copyin` and `copyout`

The `copyin` and `copyout` family of functions provide safe access to user process memory for the kernel. These are used by system call code to extract fixed-size blocks (`copyin`) or null-terminated strings (`copyinstr`) from the current user process, or insert fixed-size blocks (`copyout`) or strings (`copyoutstr`) into the current user process. These must avoid crashing the kernel if handed invalid or unaligned pointers and must not allow user programs to inspect kernel data by passing addresses that belong to the kernel. This interface is the same in Goldfish, OS/161, NetBSD, FreeBSD, and essentially all other Unix-type systems, except for Linux. (Even there, the same operations exist, just under different names.)

These functions can be implemented in three ways. On machines where the kernel is mapped into the address space of every user process, the copies can be done with ordinary memory access instructions. Machine-independent C versions of these functions are provided for this case. They must clip the region being copied to the bounds of the user-space portion of the address space, so user programs cannot leverage them to access kernel memory. On some machines (such as x86 versions with the SMAP, supervisor mode access protection, feature, and also RISC-V) they must also toggle mode bits in the processor before and after the copy. In addition, they avoid crashing the kernel on bad pointers by setting a hook in the machine-dependent part of the thread structure. The (C-level) trap dispatching code uses this hook to jump to a recovery routine if a fatal kernel trap occurs. In this implementation, the recovery routine is `longjmp`, which is used to return to a `setjmp` call at the beginning of the copy operation and return failure.

On machines where the kernel address space and user process address spaces are disjoint, special instructions must be used to access user process memory. On these machines the `copyin` and `copyout` family must be generated at least partially in assembler. Because Capstan cannot generate loops, this is done by looping in C and calling assembly code to fetch or store each word or byte. The same `setjmp` and `longjmp` mechanism is used to recover from fatal traps caused by bad pointers. (This assumes that the special instructions trap normally on error, as one would expect so that page faults can be handled. However, the scheme in Goldfish is set up to allow these assembly entry points to fail instead in case that is needed.)

As with other specification refinements, the choices are defined in a machine-independent Tuna file, one is chosen by a machine-dependent Tuna file, and matching in the Grouper front end and in Hermitcrab templates is used to generate the proper code for each alternative. This includes adjusting the machine-dependent kernel configuration to include the sources for the C version when needed, adding the necessary hook (and `jmp_buf` for `setjmp/longjmp`) to the machine-dependent thread structure as described in Section 8.3.7 above, and also producing the code itself.

As of this writing these specifications are still a work in progress. Moreover, currently both the Cassiopea language and Capstan lack the hooks necessary to define instructions that access alternate address spaces, so the synthesis problems cannot be handled yet. Also, the hooks for SMAP have not been implemented.

TLB operations

The TLB-related operations allow the machine-independent virtual memory system to invalidate memory translations. On machines with hardware-refill TLB (such as x86 and RISC-V), it is not possible to know which translations have been loaded into the TLB, so any time a mapping is removed it must also be flushed from the TLB. (Including in some cases on more than one CPU, which is known as "TLB shootdown" – the TLB shootdown logic is also machine independent, using a portable model of inter-processor interrupts, which appears in platform-dependent rather than machine-dependent code.) The TLB operations are:

- `mmu_invalidate`
- `mmu_setas`
- `mmu_make_pte`

The `mmu_invalidate` function removes a single page translation from the TLB by its virtual address, on the current CPU. The `mmu_setas` function changes the currently active address space in the MMU. Depending on the machine this may flush the entire TLB. It also may need to load the base address of the new address space's page tables into a control register, or in some cases into a global variable for access by a software TLB refill handler. The `mmu_make_pte` constructs a page table entry from a physical address and a set of machine-independent page protection flags. The mmu_setas function loosely corresponds to pmap_activate in NetBSD. The other operations are much simpler and lower-level than any of the BSD pmap interfaces.

These specifications are also still a work in progress. The TLB update operations require a lightweight TLB model, much like the cache model. This has been left to future work.

The refill handler for machines with software-refill TLB is discussed in Section 8.3.8.

The physical page operations are:

- ppage_zero
- ppage_copy
- ppage_map_swap
- ppage_unmap_swap
- mmu_map_kpages
- mmu_find_kpages
- mmu_unmap_kpages
- mmu_map_ptpage
- mmu_unmap_ptpage
- mmu_map_coremap

The various map and unmap operations create temporary mappings for different kinds of pages (swap pages, kernel heap pages, page table pages) on machines where it is impractical or impossible to map all of physical memory in the kernel's address space. The mmu_map_coremap function is similar, except it creates a permanent mapping for the coremap (the root of the reverse mapping structure). The mmu_find_kpages looks up the physical address of kernel heap pages so they can be freed in the machine-independent page tracking. The ppage_zero and ppage_copy functions zero and copy physical pages when needed. Currently machines where all of physical memory cannot be mapped into the kernel's address space are not supported (this is left for future work), so all these functions have simple default implementations.

The ppage_zero and ppage_copy functions are comparable to the pmap_zero_page and pmap_copy_page functions in NetBSD. The mmu_find_kpages function is a special case of pmap_extract in NetBSD. Otherwise NetBSD's pmap functions are much more complicated.

In addition to the above, there is a rather large Hermitcrab specification for a header file vm.h, which contains a range of definitions for parameterizing the machine-independent virtual memory code. This includes, for example, the number of layers of page table and the properties of each.

Discussion

Note that this design is aggressive and has not yet been validated against a large number of architectures.

I have not made any effort to lift design limitations of the OS/161 virtual memory system, which in this context means most notably (a) that physical memory is assumed to be one continuous block, (b) there is no support for address-space IDs (instead, everything is flushed on every change of address space), (c) there is no support for superpages, and (d) there is no support for sharing pages between processes, either by copy-on-write fork or by memory-mapping files. Adding support for these (particularly superpages) would complicate the interface some.

Establishing the range of available physical memory is not part of this interface, because it is both platform or bus/mainboard dependent (rather than machine-dependent), and also because it is part of kernel startup, discussed next.

8.3.14 Kernel Startup Code

The kernel startup code, like some trap handling, can be *machine-specific* rather than merely machine-dependent. Parts of it also may be *platform-dependent* (in the bus/mainboard sense of "platform"). The kernel startup code has a simple interface: it is invoked when the kernel is first loaded, and it calls the kernel's `main` function (`kmain` in OS/161 and Goldfish, `main` in BSD) as soon as possible, after which a machine-independent sequence of initializations takes over. Unfortunately, the things it must do are fraught with machine-dependent complications.

The core requirement for the kernel startup code is that it must get the machine into a state fit for executing C code. Some aspects of this are machine-independent concepts (e.g., it must set up a stack for the C code) and some are partly machine-independent concepts (on many machines, the kernel startup code must create initial page tables for the kernel) and others are entirely machine-specific. On x86, for example, the kernel startup code must fill out a number of tables that configure the processor and update the processor state to point at them. Some of these correspond loosely to concepts in other processors (such as control registers that hold the addresses of exception handlers) but others, particularly the segment tables, do not correspond to anything at all elsewhere and are completely x86-specific. To add to the challenge, while the kernel startup code is the first thing that runs in the kernel, the state of the machine at that point is whatever the bootloader or firmware left behind, including the types and locations of any boot arguments, which may vary from platform to platform even for the same machine architecture.

This makes dividing the startup code into machine-independent steps (so as to use Grouper) effectively impossible. While it would be possible to treat the whole sequence as a single block, and then use predicate abstraction to create a notionally machine-independent specification for that block, it would be far too large to synthesize. OS/161's kernel startup code for MIPS on System/161 is comparatively simple, but it is over an order of magnitude larger than the current synthesis scaling limit, and includes (for TLB initialization) a loop that our synthesis engine cannot handle. Kernel startup code for x86 would be far larger.

I believe the correct solution to this problem, at least in the near term, is to write per-machine Grouper specifications. Writing specifications as part of doing a port is disappointing, relative to being able to just write a machine description, but it remains preferable to writing the code by hand. (Specifications are inherently easier and faster to write than code, even in these circumstances where multiple iterations might be needed to settle on a specification where synthesis finishes in a reasonable amount of time. To write a specification one needs only to determine *what* needs to happen, and the tools take care of *how*. Among other things, this avoids the need to learn the instruction set in detail. For example, there is no step in generating `cpu_idle`, as seen in Figures 7.1 and 7.2 in which the porter has to figure out what instructions can be used to change the interrupt state.) Goldfish is not currently set up to handle machine-dependent specifications so I have left this for future work.

There is a second entry point in the kernel startup code shown in Figure 8.1. It is the entry point for starting up secondary CPUs, called `cpu_start_secondary`. It must do most of the same things that the main startup code path does with the boot CPU. This too can become mainboard- or SoC-specific as boards (and multicore processor models) vary in how secondary CPUs or cores are started. In some cases they are disabled until the kernel is ready to start them; in others they all start executing the main startup code at once in parallel and that code must choose a boot CPU and have the others wait in a loop until the kernel is ready. The `cpu_start_secondary` entry point is for use by bus/mainboard code that needs to (for example) provide an entry point to firmware. I have not changed this interface from OS/161 to Goldfish. In NetBSD the interface is much the same in broad outline, but varies in detail from machine to machine. For example, in the MIPS port the function `cpu_trampoline` serves much the same purpose, and has been abstracted some so it can

	MIPS	RISC-V	
File	lines	lines	Description
`regs.tuna`	34	32	Declares general-purpose registers.
`controlregs.tuna`	7	8	Declares control registers.
`basic.tuna`	7	7	Fundamental machine properties (such as the name).
`types.tuna`	16	16	Definitions for C types.
`asmdefs.tuna`	59	46	Assembly language templates and canned instructions.
`cachedefs.tuna`	6	6	Cache geometry definitions.
`grouperdefs.tuna`	17	17	Grouper machine description. (Section 7.3.4)
`machine.grydesc`	153	137	Grayling machine description. (Chapter 6)
Other	35[1]	0	
Total	334	269	

Figure 8.4: *Goldfish machine description elements. Line counts were taken with `wc -l` after removing comments and blank lines. These are used for expressing specifications and generating synthesis problems.*

[1] *The MIPS-specific file `regdefs.h`.*

be shared by model-specific startup code for two different MIPS models that work quite differently.

8.4 Describing Machines

Writing a new port for Goldfish (or at least the parts of it that this work handles) means writing a new machine description and connecting that machine description to the OS's specifications. For the purposes of this discussion, I shall distinguish *machine description* files, which reflect properties of the machine alone, from *lowering files*, which provide values for OS-specific abstract values and predicates used in the OS's specifications. (We call the process of substituting machine-dependent values into a machine-independent specification "lowering", because it transforms a higher-level specification to a lower-level one.) This distinction is helpful for understanding the purposes of the various per-port materials and elements. Note however that it is also somewhat artificial, because in most cases the specific OS-independent properties of the machine that need to be defined (and the names they appear under) also arise from the OS's specifications and are thus themselves OS-dependent.

8.4.1 Machine Description

The machine description files in a Goldfish port are shown in Figure 8.4. These are the descriptions used for writing specifications and producing synthesis problems.

The file `types.tuna` gives concrete C types for abstract C types, such as `unsigned long` for `size_t`. Most of these types come from the C standard; the rest are either POSIX or are (like

	MIPS	RISC-V	
File	lines	lines	Description
`machbase.casp`	239	330	The four most useful general-purpose registers.
`machmisc.casp`	245	334	Six general-purpose registers and the stack pointer.
`machgeneral.casp`	291	382	All general-purpose registers.
`machktrap.casp`	327	538	Specific registers for trap handling.
`machctl.casp`	328	565	Base plus non-MMU control registers.
`machfull.casp`	478	601	Everything.

Figure 8.5: *Cassiopea machine description variants. Line counts were taken with* `wc -l` *after removing comments and blank lines. These are used for synthesis. Each variant includes a different subset of the machine, to speed up synthesis. (See text.)* `machfull.casp` *is the union over all the others and can be thought of as the "true" size.*

`vaddr_t`) common among Unix-style kernels. Note that while in Goldfish these are specified by hand, most or all could in principle be deduced automatically from the compiler using standard techniques seen in GNU `autoconf`. The file `asmdefs.tuna` provides templates for assembly language syntax. It also includes canned instruction sequences for basic operations not covered by synthesis, such as procedure return, unconditional jump, and triggering a system call. The file `grouperdefs.tuna` (the Grouper machine description) references several of the other files, so the line count shown in the figure includes only the definitions unique to it.

None of these files (except for `regs.tuna`) is *really* OS-independent. A setup for another OS would, however, need much the same information in a similar layout. (As discussed in Section 8.3.12 the cache description would probably be more elaborate so as to be less static. For example, currently the cache line size is always a constant; it should be optionally a variable probed at boot time.)

Ideally there would be only one Cassiopea machine description per machine and this would always be used for synthesis. Instead there are six variants, with different sets of registers enabled. (Some instructions are also affected; for example, on MIPS the TLB handling instructions can only be enabled if the TLB-related control registers are enabled.) These are shown in Figure 8.5. Because the number of registers greatly affects synthesis performance, using smaller subsets where possible considerably improves overall scalability. The collections of registers chosen are an arbitrary tradeoff between maintenance overhead, resulting operating efficiency, and the desire to keep their definitions machine-independent. All descriptions include the always-zero register on machines that have one (which includes both MIPS and RISC-V). The "general" description excludes registers in general-purpose register file that are reserved by convention for special purposes (such as `k0`/`k1` on MIPS). The "ktrap" description contains the specific set of registers needed to generate blocks early in the trap handling sequence, or late in the trap return sequence, including any relevant control registers.

File	MIPS lines	RISC-V lines	Description
`*.mk`	4	2	Build glue, not very interesting.
`builddefs.tuna`	4	4	Definitions for the OS build system.
`kerndefs.tuna`	18	16	Core kernel elements.
`lockdefs.tuna`	13	9	Canned code blocks for locks and memory barriers.
`scabi.tuna`	22	22	Definitions about the system call ABI.
`trapdefs.tuna`	6	6	Definitions related to trap handling.
`userdefs.tuna`	8	8	Definitions for executing in user mode.
`vmdefs.tuna`	57	48	Virtual memory system definitions and parameterization.
`hacks.tuna`	6	6	Things I would rather were unneeded.
`mappings.casp`	11	11	Lowering definitions for Alewife.
Total	149	132	

Figure 8.6: *Goldfish lowering file elements. Line counts were taken with* `wc -l` *after removing comments and blank lines. These do not necessarily include all the needed lowering material for the incomplete specification groups (timers, traps, cache control, and virtual memory) though they include some of it.*

On MIPS this includes `k0` and `k1` and the stack pointer, but no other general-purpose registers. Control registers not required by *any* specification are not enabled even in `machfull.casp`. One should think of the size of `machfull.casp` as the "true" size of the synthesis machine description.

It is also critical to note how the total size of the MIPS machine description material (around 800 lines) and the lowering material (about another 150 lines so far) compares to the size of the original OS/161 MIPS code (around 2600 lines) and the handwritten version of the Goldfish MIPS code (around 1500 lines). In the initial Cassiopea and Capstan work [16] there was some concern about whether the mapping material would turn out to be larger than handwritten code. These concerns do not seem to be borne out when deployed in a system rather than for a series of standalone examples.

8.4.2 Lowering

Such concerns also do not reflect the amount of effort required to produce these lines. The bulk of the machine description material (in fact of all the material for a port) is the Cassiopea description, which is mostly instruction descriptions. There is some reason to hope that in the future full formal machine descriptions will come from processor vendors (as discussed in Chapter 1) and that applied descriptions like ours will be automatically derivable. Even without that, however, our experience on this project (among several people writing Cassiopea machine descriptions for a number of machines) is that it typically takes about a day to write and mostly debug one, even with little or no prior familiarity with the machine. It is unlikely that anyone can write a comparable fraction of an OS port by hand in one day, even for Goldfish with its relatively small machine-dependent interface. I would certainly not claim to be able to do it. Meanwhile, most of the rest of the machine description material is straightforward transcription from the architecture manual. The Grayling description and Tuna description files for RISC-V took only a few hours to write.

The lowering files in a Goldfish port are shown in Figure 8.6. The file `builddefs.tuna` allows setting per-machine compiler and linker flags. (For example, on MIPS kernels are usually built as non-position-independent code, which is normally not the toolchain default and requires extra compiler flags.) `hacks.tuna` mostly contains choices for explicit scratch registers. The lowering definitions in `mappings.casp` are used for the system call stubs.

Note that the total size of the lowering files is quite small. These lines are harder to write than the machine description files, as they require not just transcribing information from the architecture manual but making decisions about how the OS should work with the machine. They are small enough, however, that it is reasonable to claim that the burden of writing them is considerably smaller than the burden of writing the corresponding code by hand. Also my subjective experience has been that they are more like filling out forms than writing code, and thus materially easier.

8.4.3 Machines in Goldfish

I have written machine descriptions and lowering definitions for the following machines:

- MIPS (Native OS/161 MIPS, roughly mips32r1)
- RISC-V (RV32I)

Everything in MIPS and nearly everything in RISC-V can be represented, but as much of the code generation is not yet working, little has been tested yet.

8.5 Generating Code

Generating code for a port involves first running Nautilus, the Grouper prepare phase, Grayling, the Grouper generate phase, and Anchor (the Alewife compiler). These are all fast. This produces approximately a hundred machine-dependent block specifications that require synthesis. After synthesis (which is not fast) one runs the Grouper collect phase and then Hermitcrab. This produces source files which can be installed into the OS source tree, compiled, and (with luck) run. There are enough moving parts that setting up the build process was not entirely trivial and took several successive attempts. The final version provides diffs against a reference version for all build products and allows substituting a handwritten dummy version downstream for any potentially broken build product. It also caches synthesis results: with Capstan verification is far faster than synthesis [16], so it makes sense to try verifying an existing synthesis result against the latest specifications and only resynthesizing if that fails. All of this is driven with BSD make.

8.5.1 Synthesis Performance

Some of the synthesis specifications are degenerate (precondition equal to postcondition) such that an empty output file is sufficient. These synthesize rapidly. The rest take more time.

Per-block synthesis performance generally ranges from less than a second for some blocks (such as the cache control operations) to 20-30 minutes. Experience with the Capstan synthesis engine has taught us that blocks that fail to synthesize in roughly this time rarely succeed even if allowed to run for days.

Because the performance of the underlying SMT solver has very high variance, synthesis performance generally has very high variance as well[5]. Benchmarking it in a robust manner so as to get significant, meaningful, and reproducible results is a nontrivial undertaking, requiring many runs per test block and explicitly permuting the solver inputs within the synthesis engine. I have not attempted this for the Goldfish synthesis problems.

Instead, with the goal of only getting a rough idea of the performance, I ran the complete available set of Goldfish synthesis blocks, roughly half the potential total across the interfaces described in this chapter, through Capstan and recorded the total elapsed time. (This includes some blocks belonging to problems for which the overall specifications are incomplete.) That time, averaged over five trials for MIPS, is slightly over 30 minutes. This was done on a not particularly fast machine (a 2.9 GHz dual-core Intel i3 with BIOS date in 2009 and only 6 GB of RAM) so would be faster on a big compute server. This result should be taken only as an indication that the total elapsed time is small enough to make the goal feasible – a time measured in hours is acceptable, a time measured in days or weeks probably not.

Note however that this is for blocks that do not time out. Including the total time for blocks that

[5] While working on Penguin, I once found that synthesis got slower by two orders of magnitude after fixing a bug that had been generating duplicate assertions.

Category	Trivial	Succeeded	Timed out	Out of memory	Unsat[1]	Total
Standard C functions	4/3	4/5				8
Low-level CPU operations	1	4				5
Kernel threads	5	6				11
User system call stubs		2				2
User startup code	2/3	6/5				8
Totals	14	20				34

Figure 8.7: *The success and failure of synthesis by specification category. The numbers shown are counts of the synthesis blocks required. (Trivial blocks are satisfiable by an empty program.) Categories where the specifications are incomplete are not shown. Where two numbers are given, they are for MIPS and RISC-V respectively. None of the blocks fail.*

[1] *Unsatisfiable in five instructions.*

time out automatically changes the time to "too long", and including *n* copies of the timeout period in the does not produce a particularly meaningful result. Currently no blocks time out; however, this might change as more ports are brought on line as the number of instructions required for a given task can vary substantially from one machine to another. I chose a timeout of 45 minutes per block, because as already noted blocks that take longer than this rarely succeed even given much more time. If for a new port large numbers of blocks start timing out, the total elapsed time to discover this could extend to days. This seems unlikely, however.

8.5.2 Synthesis Success

So far none of the blocks have proved to not be synthesizable. Blocks can fail either due to running out of memory, timing out, or not being satisfiable in the maximum number of instructions Capstan considers itself able to handle.

Blocks that are too big to synthesize can always be subdivided, but subdividing reaches a limit where the steps cease to be even notionally machine-independent. Going past that point serves no purpose, either for this work or in general. Fortunately, this has not been necessary.

I have, for each machine, hand-written kernel startup code, and machine-specific trap handlers as needed, so as to be able to build and test a complete system.

Many of the blocks are trivial, that is, are satisfied by the empty program. There are two main sources of these: many are blocks inserted adjacent to call, linkage, or context blocks where ultimately nothing needs to be done. The rest generally arise when a machine-independent step in a procedure reflects considerations that do not apply to the target procedure. (For example, the interrupt enabling hooks in `cpu_idle`, described in Section 8.3.4, are only needed on some machines; on others the

	Handwritten lines for MIPS:			Generated lines for MIPS:		
Category	C	Asm	Other	C	Asm	Other
Basic machine properties	36	-	-	22	-	-
Build configuration	-	-	79	-	-	85
Standard C functions	5	41	-	5	54	-
Low-level CPU operations	175[1]	-	-	53	73	-
Timers	25[1]	-	-	-	*incomplete*	-
Kernel threads	54	49	-	55	70	-
Traps	60[2]	229	-	51[2]	*incomplete*	-
User system call stubs	-	28	-	-	30	-
System call ABI definitions	21	-	-	21	-	-
User startup code	-	23	-	-	36	-
Cache control	11	31[3]	-	12	*incomplete*	-
Virtual memory	163	101	-	69	*incomplete*	-
Totals	755	502	79	288	263	85

Figure 8.8: *Sizes of handwritten and generated code for MIPS. (The generated sizes for other architectures are not substantially different.)*

[1] *Includes handwritten assembler.*

[2] *Excludes* `trap.c`*, which is not synthesized.*

[3] *Stubbed out in the handwritten code; a real implementation would be larger.*

Note when comparing to Figure 8.1 that in addition to `trap.c`*, the rows for CPU identification and kernel startup code have been dropped.*

code blocks for them become empty.)

A chart showing the number of trivial blocks (and the number of failures, which is zero) appears in Figure 8.7.

8.5.3 Output Size

Figure 8.8 gives an overview of the generated code sizes for MIPS, in comparison to the handwritten code sizes. They are largely comparable, particularly the C code. For the C code this is because it is templated and the templates are based on the original handwritten versions. For the assembly code it is because there is limited scope with small synthesis blocks for the synthesis engine to find clever shorter implementations. To some extent this is also because the MIPS architecture does not lend itself to such implementations. (Recall from Chapter 3 that the synthesis engine will, in its default mode at least, always pick the shortest implementation it can find. So in general we expect the synthesis output to be smaller than handwritten code. Exceptions occur primarily when there are instructions the synthesis engine does not know about that happen to fit the problem perfectly. This possibility is one of the motivations for the future peephole optimizer proposed in Section 5.8.)

Category	MIPS	RISC-V
Basic machine properties[1]	✓	(✓)
Build configuration[1]	✓	(✓)
Standard C functions	✓	(✓)
Low-level CPU operations	✓	(✓)
Timers	*specifications incomplete*	
Kernel threads	✓	(✓)
Traps	*specifications incomplete*	
User system call stubs	✓	(✓)
System call ABI definitions	✓	(✓)
User startup code	✓	(✓)
Cache control	*specifications incomplete*	
Virtual memory	*specifications incomplete*	

Figure 8.9: *Overall results for each category of generated code. Code that has been successfully generated and tested appears with a check mark. Code that has been successfully generated and* seems *correct appears with a parenthesized check mark.*

[1] *These elements are tested by compiling the code.*

8.6 Testing the Code

While the synthesis engine always produces code that meets its specifications (if it succeeds), proving code correct does not mean that it will necessarily work; the specification could be wrong, or there could be bugs in the machine description so that verification approves bad code, or there could be bugs elsewhere in the system.

Therefore, it is desirable to also test the synthesized code. Since testing the code requires booting the Goldfish kernel, considerable engineering effort beyond the synthesis runs is needed to handle the code our project does not cover: device drivers and mainboard/bus code. For MIPS, or at least MIPS on System/161, this code already exists, as it was written as part of OS/161. For other platforms it did not, and consequently full testing for all platforms is a large future project.

For RISC-V I plan in the future to create a version of System/161 with RISC-V instead of MIPS CPUs. This requires no new device drivers and only minor changes to the mainboard and bus code.

8.7 Summary of Results

Figure 8.9 contains an overall summary of the results. For four of the categories the specifications are not complete. I can generate some code for these, but not all of it. For the other eight the generation is complete. All of the MIPS code has been tested (it produces a working kernel), but the RISC-V code has not yet been tested.

Chapter 9

Discussion and Conclusions

This chapter discusses some broader points before concluding.

9.1 Machine Assumptions

Overall I have attempted to avoid making any readily avoidable assumptions about the target machine architectures, to maximize the applicability of the tools (and the results). The reader may have noticed that Grayling can be told the bit size of bytes. It is easy and costs little to carry that around as a parameter and apply it when converting between bit sizes and byte (memory cell) sizes, if done when initially writing the code. It is much harder to refit such generality afterwards. In the case of Grayling I expect (but have not checked) that if you tell it to use 36-bit bytes and write a suitable machine description it will be able to generate PDP-10 code.

Other generality is more expensive, so I did not invest effort on support for dead or exotic architectures or even architectures that differ in inconvenient ways from the mainstream. I already observed in Section 7.6.2 that Grouper is limited to register machines, because working with values in registers is a natural approach and consistent with mainstream machine architectures. Grouper currently draws more distinctions than necessary between registers and memory cells such that some specifications are unnecessarily difficult to write; for example, one cannot have a value mentioned explicitly in a machine-independent specification be in memory on one machine and in a register in another. This could be improved, which would help in handling register-starved machines such as 32-bit x86; but doing so would still not make Grouper work well with stack machines.

In the case of Goldfish, every OS inherently comes with a model of what machine properties are universal and what machines can be supported. Changing this usually requires revisions across the entire machine-independent code base and can be extremely expensive. I have not attempted to change the assumptions that OS/161 makes about machines. It has always been intended to be portable to set a good example for students, but it is probably not 64-bit clean (originally I ignored this and it has never explicitly been rectified) let alone capable of being run on a PDP-10.

OS/161 and thus Goldfish are also inherently limited to machines and platforms that are intended to run Unix-like operating systems; for example, the machine must have an MMU, support access to external hardware devices that can generate interrupts, allow at least 1 MB or so of RAM, and have a word size of at least 32 bits. Neither will ever run directly on a GPU, for example; it would not make sense to try.

9.2 Concurrency

As mentioned Chapter 1 machine vendors are increasingly involved in producing formal specifications for their machines, and in the future we expect that applied descriptions like our Cassiopea ones will be automatically derivable from vendor-provided material.

In this world the power-to-weight ratio for synthesizing locking and memory barrier operations becomes less adverse. Given that mistakes in concurrency code also notoriously lead to bizarre untraceable problems, synthesizing it is in general desirable. It is possible that this requires a complete treatment of the processor's memory model. This is not necessarily prohibitive, as Memsynth [5] successfully feeds one to a solver. It is also possible (especially for memory barriers) that there is a lightweight model waiting to be invented. Looking for such a model would be an interesting project for future work. Looking further forward, the use of synthesis for concurrent code might also make relaxed cache consistency models practically feasible.

Meanwhile, it would be interesting to integrate existing work that can synthesize code for lock-free data algorithms as another block generator for Grouper. As in the current model where the concurrent blocks are canned, the rest of the code (all the blocks that do not involve concurrent accesses) would not need to reason about concurrency.

9.3 Synthesis vs. Coding From Specs

The work for this dissertation has really in some ways done two separable things: first, divided the problem of porting an operating system into bite-size chunks, and then using program synthesis and other techniques to generate those chunks. This observation immediately gives rise to two questions: first, what is the relative importance of these two developments? And second, given the bite-size chunks, is program synthesis (which remains experimental and not ready for production) really the best way to proceed? One could hand-write a port one chunk at a time; is that easier or harder than writing the machine description and lowering material needed to drive synthesis? After all, if the chunks are 1-3 instructions (And is it easier or harder than writing a port entirely by hand?)

There are several points to raise in this context. First, Goldfish has been engineered to avoid much unnecessary machine-dependent code; this is an important development, but does not affect these questions. Such work makes Goldfish easier to port in an absolute sense: it is true whether or not the code is written by hand.

Second, however, though Goldfish may have boiled the problem down closer to its essentials, those essentials still remain. The system needs kernel thread switch code; it must be produced somehow. One of the important things about the synthesis approach is that, even though some parts of the port still (for now) require explicit design choices, many do not. The kernel thread switch is a good example: using the Aquarium approach, it disappears entirely. The Grouper and Grayling specifications together depend only on three things besides core machine properties: global choices of two scratch registers, and the identity of any current thread or current CPU register chosen as part of a design decision for a different porting problem. (And of those, the scratch registers are only required because of an easily overcome weakness in the current Grouper code.) This means that if generating a port by synthesis the porter need not think about kernel thread switch at all, any more than they need think about file system code. Writing a port by hand, regardless of whether this is done in small chunks or not, still requires writing and thus thinking about the thread switch code.

Third, dividing the assembly language coding problem into chunks of 1-3 instructions does not automatically make it easy. One may think of writing small blocks of assembler as easy and fast; but that is only true if one already knows the machine. After years of teaching with OS/161 and MIPS (and having implemented System/161, and Penguin, and other things for MIPS) I could probably at this point write much of a MIPS port of a system like Goldfish without even opening the architecture manual, and it would be quick and fast whether or not the problem was divided into chunks;

but this is not true of machines I have never worked with, such as PowerPC. A one-instruction block where the specification required writing a register to memory (such as some of the blocks in `crt0.S`) would require a search through the manual to find an instruction, and then likely another to figure out what the addressing modes in the instruction description mean, and perhaps more, and this would need to be repeated until I learned the architecture. Meanwhile, writing in very small chunks creates ergonomic issues: the size of the chunks destroys high-level structure and the work would likely be tedious and repetitive, but not similar enough from one block to the next to give any respite from the difficulty so also likely error-prone. One might call it "assembly line coding". Even if one can verify the output (as mentioned at the end of Section 7.7) it is not necessarily an effective approach. Moreover, the sheer number of blocks required poses a problem. The original motivation for developing Grouper was that writing the specifications for these blocks one at a time was prohibitive and involved too many pieces to manage by hand. It seems unlikely that writing the code in the same pieces will fare much better.

Fourth, as we saw in Sections 8.4.1 and 8.4.2, the total size of machine description and lowering files is considerably less than the size of handwritten code, so even assuming that the difficulty of producing each is comparable, and even without being able to get some or all of the machine description material from the manufacturer, the synthesis approach wins. Moreover, the difficulty is not comparable. As I already mentioned (in Section 8.4.1) our experience is that a Cassiopea machine description takes about a day to write, and my experience so far is that the Grouper and Grayling descriptions are much easier and faster, and that lowering material (both for Goldfish and other examples) is faster still. The design decisions needed for some lowering definitions (such as the method for tracking the current thread) can often be made from the high-level overview that appears in the introductory parts of architecture manuals without getting involved in details at all, and many of the others are a matter of finding the right place in the manual and then transcribing.

Finally, in the long run, computer time is cheaper than human time. Production grade assembly synthesis (for blocks of the size used herein) is at this point available for at most a few man-years of engineering work, if anybody cares to spend them. In this light the synthesis approach would have to be clearly inferior to not be preferable in the long run, and even if the prior arguments are not fully convincing to every reader the available information does not support the *opposite* conclusion.

Applying these points to answer the original questions in turn: first, while I think compositional code generation is probably the most important contribution of this dissertation, it is important because it enables automated reasoning and code generation. It is not clear that dividing up a well-specified small problem (of the scale of the kernel thread switch) into smaller pieces before handing to a human programmer offers any benefit. Doing that subdivision is one of the first things people learn as beginning coders. Thus I will say that dividing the problem of porting into very small chunks is not as important as being able to process many or most of them automatically from simpler declarations. And I will furthermore say that generating these chunks is preferable to writing them by hand.

(Generating them by synthesis might or might not be the best way to proceed; it is the way we chose as a general-purpose solution for a wide range of problems. I have already mentioned the possibility of other block generators; compiling from a machine-independent general purpose language designed to work with Grouper, or more than one such, already mentioned in passing in Sections 1.1.1 and 7.7, is a particularly attractive alternate or additional approach.)

9.4 Conclusions

The primary broader question regarding all this material is: "Is this a good or promising approach?" It is not a complete solution, but is it part of one? (The dream, of course, is to download a machine description, push a button, come back maybe a few days later, and have an OS ready to go.) Speaking with my kernel developer hat on, I think there are several parts to this question.

First: is synthesis a good way to create some parts of the machine-dependent code of an operating system? Second: does this hold for larger procedures, for example, those that require multiple blocks composed with Grouper? Third: is synthesis a good approach for machine-*specific* procedures?

The answer to the first question is an unqualified yes. Even though our synthesis engine is a research artefact and not very mature, it produces correct code for small instruction sequences that do specific things, such as flushing a particular cache line. This is important, because mistakes in such sequences are easy to make, not easy to find, and tend to lead to difficult problems. For example, if you fumble the magic number that specifies the cache operation so that you do a discarding flush (just invalidate) rather than a write-back flush (write-back-and-invalidate) prior to DMA-based I/O, you get corrupted data... occasionally, depending on how often the data is dirty in the cache. Diagnosing problems of this kind is largely impossible.

If the code is synthesized from a machine-independent specification, the specification will (in most cases) have already been debugged when previous ports were built. Meanwhile, even if not, or if errors in the machine description lead to synthesizing the wrong code, inspecting the output assembly offers an independent cross-check. That cross-check, meanwhile, requires less familiarity with the machine than writing the code directly does; it amounts to recognition vocabulary vs. active vocabulary. So synthesis beats verification of handwritten code.

The chief drawback of synthesis is that the output code can be bizarre and/or unmaintainable. While we have seen this (for example, choosing formally correct but inappropriate registers to hold temporary values) the principle that state not explicitly modified by a postcondition (or declared as scratch) mostly prevents this. Furthermore, sequences of instructions we can generate are not large and do not readily become incomprehensible.

The previous discussion applies to small procedures. Is it still true for larger procedures, e.g., those that require multiple blocks composed with Grouper? The answer here is less immediately clear, as Grouper is a research prototype and has limitations that could likely (or certainly) be removed with more engineering effort. For example, it is currently difficult to implement a machine-dependent interface using subprocedures whose existence or specification is not machine-independent. This is a tooling problem and not a fundamental limitation; one can easily have blocks within a procedure that are empty and do nothing on some machines and not others. My conclusion is that the approach shows promise, but it is too early to tell for certain. Grouper is set up so that synthesis runs offline, in the sense that the synthesis engine and Grouper are not running simultaneously. This was necessary for development (for example, to be able to easily inspect the specifications being sent to the synthesis engine), but because it requires Grouper to generate a predictable set of output synthesis problems, it hampers Grouper's ability to adjust to machine properties on the fly. The original vision included, for example, the ability to allocate scratch registers, and to spill registers to the stack when needed (and when permissible). The latter requires the ability to insert additional blocks into the specification on demand, and thus produce additional synthesis problems depending on circumstances. Spilling registers well also requires the ability to update the specification on the fly based on the results of synthesis (to, for example, decide to spill based on how many scratch registers the synthesis engine says a particular block needs) and this does not work at all with the offline synthesis scheme. Meanwhile, while I stand behind OS/161 as an experimental platform deployment in a production OS will require tackling a variety of issues that were intentionally simplified out of OS/161.

The third question is: "Is synthesis a good approach for machine-*specific* procedures?" While this is not something I have experimented with directly, I believe the answer to be yes, for the following reason: writing specifications is easier than writing code. Generating the kernel startup code for 32-bit x86 (as an extreme example) will require writing specifications for machine-specific steps such as initializing the segment descriptor tables; but, speaking from the perspective of having written that code once by hand many years ago, if I were doing the x86 port of a new operating system I would far rather write the specifications.

There is one other conclusion that I can definitely draw, and that is that the Grouper model of generating code with different tools on a per-block basis is a good one. For justification, consider Grayling: it is perhaps overengineered and more complicated than strictly necessary; but it is far simpler than any “real” compiler. This remains true whether or not a synthesis engine remains one of the components. One could imagine a collection of special-purpose compilation tools, each doing one specific thing well, that could be combined using Grouper or a Grouper-like tool to generate procedures that are traditionally written in assembly and resistant to being compiled.

References

[1] R. Achemann, L. Humbel, D. Cock, and T. Roscoe. Physical addressing on real hardware in Isabelle/HOL. In *Proceedings of the 9th International Conference on Interactive Theorem Proving (ITP '18)*, July 2018.

[2] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. Krishnaswami, and P. Sewell. ISA semantics for ARMv8-a, RISC-V, and CHERI-MIPS. In *Proceedings of the 46th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '19)*, 2019.

[3] G. H. Blindell. *Instruction Selection: Principles, Methods, and Applications*. Springer, 2016.

[4] R. Bodik, S. Chandra, J. Galenson, D. Kimelman, N. Tung, S. Barman, and C. Rodarmor. Programming with angelic nondeterminism. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*, pages 339–352, New York, NY, USA, 2010. ACM.

[5] J. Bornholt and E. Torlak. Synthesizing memory models from framework sketches and litmus tests. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI '17)*, 2017.

[6] R. Brummayer and A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '09): part of the Joint European Conferences on Theory and Practice of Software (ETAPS '09)*, pages 174–177, Berlin, Heidelberg, 2009. Springer-Verlag.

[7] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08): part of the Joint European Conferences on Theory and Practice of Software (ETAPS '08)*, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] J. Dias and N. Ramsey. Automatically generating instruction selectors using declarative machine descriptions. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*, pages 403–416, 2010.

[9] J. K. Feser, S. Chaudhuri, and I. Dillig. Synthesizing data structure transformations from input-output examples. *SIGPLAN Not.*, 50(6):229–239, June 2015.

[10] GNU GRUB. `https://www.gnu.org/software/grub/`.

[11] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*, pages 317–330, New York, NY, USA, 2011. ACM.

[12] J. L. Hennessy and D. A. Patterson. A new golden age for computer architecture. *Commun. ACM*, 62(2):48–60, Jan. 2019.

[13] D. A. Holland, J. Hu, E. Lu, M. Kawaguchi, S. Chong, and M. I. Seltzer. Aquarium: Cassiopea and Alewife languages. Technical report, October 2019.

[14] D. A. Holland, A. T. Lim, and M. I. Seltzer. A new instructional operating system. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education (SIGCSE '02)*, pages 111–115, New York, NY, USA, 2002. ACM.

[15] J. Hu, E. Lu, D. A. Holland, M. Kawaguchi, S. Chong, and M. I. Seltzer. Trials and tribulations in synthesizing operating systems. In *Proceedings of the Programming Languages and Operating Systems Workshop (PLOS 2019)*, August 2019.

[16] J. Hu, E. Lu, D. A. Holland, M. Kawaguchi, S. Chong, and M. I. Seltzer. Towards porting operating systems with program synthesis, 2020. Under submission to the 2020 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '20).

[17] L. Humbel, R. Achemann, D. Cock, and T. Roscoe. Towards correct-by-construction interrupt routing on real hardware. In *9th Workshop on Programming Languages and Operating Systems (PLOS 2017)*, October 2017.

[18] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: A tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th USENIX Conference on Operating Systems Design & Implementation (OSDI '04)*, pages 20–20, Berkeley, CA, USA, 2004. USENIX Association.

[19] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security '18)*, pages 973–990, Baltimore, MD, Aug. 2018. USENIX Association.

[20] P. M. Phothilimthana, A. Thakur, R. Bodik, and D. Dhurjati. Greenthumb: Superoptimizer construction framework. In *Proceedings of the 25th International Conference on Compiler Construction (CC '16)*, pages 261–262, New York, NY, USA, 2016. ACM.

[21] N. Polikarpova and I. Sergey. Structuring the synthesis of heap-manipulating programs. In *Proceedings of the 46th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '19)*, 2019.

[22] O. Polozov and S. Gulwani. Flashmeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '15)*, pages 107–126, October 2015.

[23] N. Ramsey and J. Dias. Resourceable, retargetable, modular instruction selection using a machine-independent, type-based tiling of low-level intermediate code. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*, pages 575–586, 2011.

[24] R. Rashid, J. Avadis Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proceedings of the 2nd Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS 1987)*. ACM, October 1987.

[25] RISC-V foundation. `https://riscv.org/`.

[26] L. Ryzhyk, P. Chubb, I. Kuz, E. Le Sueur, and G. Heiser. Automatic device driver synthesis with termite. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*, pages 73–86, New York, NY, USA, 2009. ACM.

[27] L. Ryzhyk, A. Walker, J. Keys, A. Legg, A. Raghunath, M. Stumm, and M. Vij. User-guided device driver synthesis. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI '14)*, pages 661–676, Berkeley, CA, USA, 2014. USENIX Association.

[28] C. Small and S. Manley. A revisitation of kernel synchronization schemes. In *Proceedings of the USENIX 1997 Annual Technical Conference (USENIX '97)*. USENIX Association, January 1997.

[29] A. Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, Berkeley, CA, USA, 2008. AAI3353225.

[30] A. Solar-Lezama, C. G. Jones, and R. Bodik. Sketching concurrent data structures. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*, pages 136–148, New York, NY, USA, 2008. ACM.

[31] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '06)*, pages 404–415, New York, NY, USA, 2006. ACM.

Appendix A

Artefacts

This dissertation comes with an archive of associated software artefacts. (You should have received it along with the text, or be able to download it from a nearby location.) This blob includes the following subtrees:

- `cassiopeia` *(sic)* – the source tree for the synthesis engine, including both Capstan itself and the Alewife compiler Anchor. It also includes a broken copy of Nautilus, which should be ignored.
- `goldfish` – the source tree for Goldfish.
- `grayling` – the source tree for Grayling.
- `grouper` – the Grouper source tree, which also contains the Tuna library, the Tuna language specification, Hermitcrab, and Nautilus.
- `penguin` – the source tree for Penguin, included for completeness.

Because Goldfish is based on OS/161 and includes OS/161 solution code, the complete tree should not be released to the general public. I have prepared a partial public version that contains at least all the material for generating ports.

A more complete roadmap and brief directions for reproducing the results in this dissertation may be found in the `doc` subdirectory of the Grouper source tree.

Appendix B

Grouper Abstract Syntax

This appendix describes the Grouper abstract syntax (to be precise, the machine-independent abstract syntax) in detail. (Note: I do not provide typing rules for the type system, or semantics for the expressions, as these formalisms would not be at all interesting.)

This appendix uses the following symbols:

w	Bit widths (small integers)
x_t	Names for types
x_r	Names of registers
x_m	Names of memory regions
x_{pp}	Names of program points
x_c	Data constructors
x_f	Names of functions
x	Ordinary variables
t	Types
p	Patterns
e	Expressions
$\hat{e}$	Separation expressions
a	Assertions
d	Declarations

$$\begin{array}{lr}
t ::= & \texttt{unit} \mid \texttt{bool} \mid \texttt{string} \mid x_t \mid w?\,\texttt{bit} \\
\mid & w?\,\texttt{bit reg} \mid w?\,\texttt{bit regset} \\
\mid & \texttt{option}\ t \mid t_1 \ldots t_n \to t
\end{array}$$

Figure B.1: *Types t in Grouper.*

$$\begin{array}{lr}
e ::= & () \mid \texttt{true} \mid \texttt{false} \mid \texttt{"abc"} \mid \texttt{0x123} \mid x_r \mid x \\
\mid & \texttt{\{}\ x_r \ldots\ \texttt{\}} \mid \texttt{[}x_m\texttt{,}\ e\texttt{]} \\
\mid & \texttt{signext}\ e\ w \mid \texttt{zeroext}\ e\ w \mid \texttt{narrow}\ e\ w \mid \texttt{sizeof}\ e \\
\mid & uop\ e \mid bop\ e_1\ e_2 \mid x_f\ e \ldots \mid x_c\ e \ldots \mid \texttt{None} \mid \texttt{Some}\ e \\
\mid & \texttt{let}\ x\ \texttt{:}\ \ t\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\
\mid & \texttt{match}\ e\ \texttt{with}\ p_1\ \texttt{:}\ \ e_1\ \texttt{;}\ p_2\ \texttt{:}\ \ e_2\ \texttt{;}\ \ldots \\
p ::= & x \mid \texttt{None} \mid \texttt{Some}\ p \mid x_c\ p_1 \ldots p_n
\end{array}$$

Figure B.2: *Expressions e and patterns p in Grouper.*

B.1 Types

Figure B.1 shows the types in Grouper. Strings may be either single-line or multi-line, like in Tuna; this appendix ignores the distinction for simplicity. x_t is an identifier bound to a type; these may be structure/record types, sum types, or alias names for other types. $w?$ indicates an optional bit width. Bit widths not given are resolved by unification during typechecking. All numeric types are bitvectors (machine integers) with some specific width. Registers and sets of registers are also tagged with their bit width. The `option` type is special-cased to allow it to be polymorphic. Functions are restricted (in both parameters and return) to types considered "base" types, which excludes `option` and register sets. Note that signedness of machine integers is not captured by the typing. This decision was made to match Cassiopea.

B.2 Expressions and Patterns

Figure B.2 gives the abstract syntax for Grouper's expressions and match patterns. Registers x_r are treated as values and are different from variables of register type. (Note that this differs from the treatment in Cassiopea.) Register sets (register names in braces) may only be literals. The form $\texttt{[}x_m\texttt{,}\ e\texttt{]}$ is a pointer, where x_m is the name of a memory region (these are not first class; see Section

$$
\begin{aligned}
\hat{e} &::= \quad \texttt{true} \mid \texttt{empty} \mid [e_1 \rightarrow e_2] \mid [x \leftarrow e_2] \mid \hat{e_1} * \hat{e_2} \mid x_f \\
a &::= \quad e \mid x_{pp} \vdash \hat{e}
\end{aligned}
$$

Figure B.3: *Assertions a (and separation expressions ê) in Grouper.*

7.2.3) and the expression gives an offset from that pointer. `signext` and `zeroext` widen bitvector values; the `narrow` construct truncates a value to the lowest w bits. (There is also concrete syntax for extracting bits and bit slices from expressions that is desugared to shifts and `narrow`.) The `sizeof` operator has two concrete forms, one for size in bytes and one for size in bits; in general both are needed and privileging either leads to confusion.

Unary operators *uop* are logical and bitwise not, and negation. Binary operators *bop* are add, subtract, multiply, divide, modulus, less-than, equal, shifts, and both logical and bitwise boolean operations, all in both signed and unsigned forms where appropriate. Functions x_f are not first class (and are not permitted to be recursive); data constructors x_c are distinguished from functions by typing. The `Some` and `None` constructors are special-cased to allow them to be polymorphic.

Patterns match data constructors, including the built-in constructors for `option` (so that `Some` can be polymorphic), plus a form to bind a value to a variable. The scope of patterns is the match case they belong to.

B.3 Assertions

Assertions are shown in Figure B.3. An assertion may be pure (or time-invariant, that is, not directly involving machine state) in which case it is just an expression, or it may involve machine state, in which case it is attached to a program point x_{pp} (the names of these come from the body declaration) and must be expressed in separation logic. The separation keys (left-hand side, e_1 in the figure) must ultimately resolve to either registers or pointers; at the machine-independent level they may be arbitrary expressions. The form $[e_1 \rightarrow e_2]$ is the conventional separation logic statement that the object named/pointed to by e_1 contains the value e_2, and the form $\hat{e_1} * \hat{e_2}$ is the conventional separation logic assertion that $\hat{e_1}$ and $\hat{e_2}$ are both true and that their keys are disjoint.

The unorthodox form $[x \leftarrow e_2]$ is pronounced "find e_2". It extracts the (or a) separation key holding the value e_2 and asserts that the variable x is equal to this key (register or pointer). (That is,

$$
\begin{array}{lr}
d ::= & \texttt{type}\ x_t = x_{c_1}\ t_1 \cdots \texttt{¦} \ldots \texttt{¦}\ x_{c_n}\ t_n \cdots \\
| & \texttt{type}\ x_t = \texttt{\{}\ x_i : t_i\ @k_i\ \cdots\ \texttt{\}} \\
| & \texttt{type}\ x_t = \texttt{type}\ t \\
| & \texttt{register}\ x_r : w = \texttt{"\$r"} \\
| & \texttt{var}\ x_r : t \\
| & \texttt{region}\ x_m : t \mid \texttt{region}\ x_m : t[k] \\
| & \texttt{region}\ x_m : t\ \texttt{with label}\ x \\
| & \texttt{region}\ x_m : t[k]\ \texttt{with label}\ x \\
| & \texttt{format}\ x_f = \texttt{"abc\$1\$2"} \\
| & \texttt{regmap}\ x_f = [x_{r_i} \rightarrow e_i : t_i,\ \ldots] \\
\mathit{import} ::= & \texttt{import}\ \mathit{dir\ file}\ x : t \\
| & \texttt{import}\ \mathit{dir\ file} *
\end{array}
$$

Figure B.4: *Declarations d and imports in Grouper.*

$[x_r \rightarrow e] * [x \leftarrow e]$ reduces to $[x_r \rightarrow e] \wedge x = x_r$.) This gives a machine-independent way to capture the identity of registers used only implicitly in the specification. It is discussed further in Section 7.4.2.

A function name x_f appearing as a separation assertion must be a function from registers (currently this does not work for pointers) to machine word values. (Generally this will be a register map, a finite map with register keys.)

B.4 Declarations and Imports

Figure B.4 gives the declarations and imports in Grouper. Not all of these are allowed in the concrete specification syntax; for example, it does not make sense to declare registers in a machine-independent specification file, so all register declarations are imported from elsewhere.

The first `type` form declares a sum type, where each constructor x_c has zero or more arguments with type given by each t_1. The second declares a structure or record type, where element x_i has type t_i and appears at offset k_i, where k is an integer constant. These are normally imported; Grouper is not equipped to compute or cross-check structure offsets in memory. The third declares a type alias. These are of limited utility directly within Grouper files but are essential for importing machine-dependent type definitions.

Declaring a register gives both a width w (in bits) and also an (optional, concretely) string form

linkage ::=	`asmlinkage` {
	`linkersym` *e*
	`globl` *e*
}	
$\mid$	`clinkage` *nautilus*
block ::=	`literal` *e*
$\mid$	`synth` *capstan*
$\mid$	`context` *grayling*
$\mid$	`call` *nautilus*
$\mid$	`clinkage` *nautilus*
capstan ::= {	
	`scratchregs` *e*...
}	
grayling ::= {	
	`specfile "abc"`
	`symbol "abc"`
	`inputs` $e_1 \ldots e_n$
	`outputs` $e_1 \ldots e_n$
}	
nautilus ::= {	
	`specfile "abc"`
	`symbol "abc"`
	`args` $e_1 \ldots e_n$
	`ret` *e*
	`retaddr` e_1 e_2
	`stackpointer` e_1 e_2
	`docalleesave`?
}	
body ::=	x_{pp}
$\mid$	*body block* x_{pp}
$\mid$	

Figure B.5: *Linkages, blocks, and bodies in Grouper.*

used if generating output to the assembler. This is useful on machines where register names are tagged with a non-identifier character, often `$` or `%`. Declaring a variable gives only a type, not a definition of the value. The value may be defined (or constrained) by making assertions about it. A memory region is defined with `region`; the region either has type *t* (normally a structure type) or *k* copies of type *t* (normally a bitvector type) arranged in a C-style array. (Arrays of structures are possible in principle but not supported by the implementation.)

`format` provides a way to use Tuna's format string feature. Importing an unsubstituted Tuna format object (which has function type in Tuna) produces one of these declarations, which declares a function x_f taking some number of strings and producing a string.

Finally, a register map declaration (`regmap`) allows importing a finite map from registers to expressions from a Tuna file. This can be used as a separation assertion, which is allows creating blocks whose exact behavior is machine-dependent.

Imports read either a specific symbol or all symbols from a given file (a tuna file) The type must agree with the imported object. The file is searched for in the directory *dir*, which is not a literal but a symbolic directory name:

- `specdir` – the specification file's directory
- `machinedir` – machine description directory
- `outputdir` – the directory for output files
- (other) – a user-defined symbolic name

The directories themselves are defined on the Grouper command line.

B.5 Procedure Bodies

Linkages, blocks, and procedure bodies appear in Figure B.5. The various fields associated with blocks are defined as follows:

- `linkersym` – a string, the linker symbol to output to the assembler as the name for this procedure. Assembly linkages only; for C linkages this is taken from the Nautilus symbol name.

- `globl` – a boolean, whether the output procedure is tagged "global" (external linkage, in C terms) or not (static linkage). Assembly linkages only; for C linkages this is reported by

Nautilus.

- `scratchregs` – registers available to Capstan in this block as scratch space.
- `specfile` – the name of the external specification file for this block.
- `symbol` – the name of this block's material in the external specification file.
- `inputs` – registers used to pass inputs to the block.
- `outputs` – registers used to receive outputs from the block.
- `args` – expressions to assert equal to the arguments (for a call) or parameters (for a linkage).
- `ret` – expression to assert equal to the return value.
- `retaddr` – entry and exit expressions to assert equal to the return address. (Linkages only.)
- `stackpointer` – entry and exit expressions to assert equal to the stack pointer value. (Linkages only.)
- `docalleesave` – if present, include assertions about preserving callee-save registers. (Linkages only.)

A procedure body is either a starting program point, or a sequence composed of a prior body and another block, followed by a program point. Control flow constructs will appear here.

B.6 Full Procedure

A full procedure specification is a list of imports, a list of declarations, a linkage, a body, and a list of assertions. A full procedure also has a name, not necessarily the same as the linker-level symbol given in the linkage, which is used to name the various output files and to reference the procedure within its specification file.

A specification is a list of full procedure specifications.