Experience Report: Writing NetBSD Sound Drivers in Haskell

A Reentrant Haskell Compiler for Operating Systems Programming

Kiwamu Okabe
METASEPI DESIGN
kiwamu@debian.or.jp

Takayuki Muranushi
The Hakubi Center for Advanced Research
Kyoto University
muranushi.takayuki.3r@kyoto-u.ac.jp

Abstract

Most strongly typed, functional programming languages are not equipped with a reentrant garbage collector. This is the reason why such languages are not used for operating systems programming, where the virtues of types are most desired. We propose use of Context-Local Heaps (CLHs) to achieve reentrancy, also increasing the speed of garbage collection. We have implemented CLHs in Ajhc, a Haskell compiler derived from jhc, rewrote some NetBSD sound drivers using Ajhc, and benchmarked them. The reentrant, faster garbage collection that CLHs provide opens the path to type-assisted operating systems programming.

Categories and Subject Descriptors D.4.0 [*Operating Systems*]: Organization and Design; D.4.2 [*Storage Management*]: Garbage collection

General Terms Languages, Design, Performance

Keywords Operating Systems, Haskell, Garbage collection, Reentrant code

1. Introduction

There are many established operating systems in use today, including Linux, OS X, and Windows. With such competition, who seriously cares about development of other operating systems? In addition to open-source operating system developers, embedded systems developers do.

Embedded systems developers program the electronic devices that control everything from printers to cars. Life would not be the same without their work. Inefficient development environments make their job difficult, however.

Embedded systems developers modify and combine device driver and operating system source code in order to create their products. Most of the source code is only available in C, and it is often outdated, unmaintained, undocumented, and unreadable to an unimaginable degree. Use of such code often results in numerous runtime errors. To make things worse, kernel and device driver errors can be difficult to reproduce. They can also be difficult to debug, as they cause the system to halt instead of raising a SEGV and passing control to a debugger. In such an environment, how is the quality of open-source operating systems maintained? The quality is maintained by the Bazaar model: "given enough eyeballs, all bugs are shallow." [9]

Embedded systems developers face tough bugs in small teams, so they are naturally attracted to the strengths of functional programming and type systems. A strongly typed, open-source operating system would detect many errors, allowing embedded systems developers to apply large changes more confidently and therefore deliver more innovative features to the products that we use.

Typed operating systems have not been developed, however, for a number of reasons. One issue is that functional programming languages tend to be highly dependent on the operating system and high-end hardware. Some software, such as microcontroller firmware, must work without an operating system. Memory managers and many drivers must run continuously in constant memory, without allocating new memory. Kernel software must handle hardware interrupts, and be reenterable while doing so. In addition, many components/systems must fit within megabytes or even kilobytes of memory, which is too small to host the runtime of many popular functional programming languages.

Reentrancy is necessary to achieve preemptive scheduling [11] in a Unix-like kernel. The definition may seem trivial: a function is reentrant if it can be *hardware* interrupted while running and safely called again from the interruption. Reentrancy is different from thread safety and referential transparency, however. For example, a function that obtains a single resource at its beginning and releases the resource at its end may be thread-safe but not reentrant. This is because the execution of the first function call is suspended by the interrupt and does not continue until the exit of the interruption (and thus the second call). The second instance will wait forever for the resource, which can only be released by the suspended first instance. If the second instance were called from another thread instead of an interruption, then it would be running in parallel and could wait for the first instance to release the resource. The function is therefore thread-safe but not reentrant.

Reentrancy may seem like an easy feature to achieve, but what about garbage collection? A hardware interrupt may arrive while the garbage collector is moving objects around and call arbitrary functions that might access the objects and trigger another instance of garbage collection! Most functional programming language runtimes would crash under such a mess.

The C programming language allows a high degree of control, but some things cannot be controlled with a functional programming language. Garbage collection is one of them. Even the most skillful programmers cannot write a reentrant function if the garbage collector is not reentrant. Given that operating systems must handle hardware interrupts, and we need reentrancy for in-

Haskell Symposium '14, September 4–5, 2014, Göteborg, Sweden.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

terrupt handlers, a reentrant garbage collector is required to implement a strongly typed operating system.

We have taken the following path to deliver a typed operating system to embedded systems developers. Since we do not have the manpower to write an entire operating system, we have adopted a *snatch* design strategy (§2), where we gradually rewrite components of an existing operating system in Haskell. In this paper, we snatch the sound drivers, as hardware drivers are representative examples of interrupt handling applications. By successfully writing these drivers, we demonstrate that our design can handle hardware interrupts.

We decided to use jhc (§3) for multiple reasons, including the fact that it produces binaries at least eight times smaller than other candidates (§3.2). We invented and implemented Context-Local Heaps (CLHs, §4) to make jhc reentrant, and we call the result *Ajhc*. We snatched NetBSD's AC'97 and HD audio sound drivers into Haskell using Ajhc (§5). The snatched drivers play sound seamlessly, while occupying a maximum of 0.6% of the CPU load (§6.3) and performing garbage collection as frequent as 33.5 times per second (§6.4). We conclude that we have opened a new path for typed operating system development.

2. Snatch: A Design Strategy

Our strategy for creating a practical operating system that is written in a strongly typed programming language is to gradually rewrite components of an existing operating system, which we call the **snatch** strategy. Using the snatch strategy, we do not have to design an operating system from scratch. Instead, we start with the existing C source code of a well-established operating system and gradually rewrite components in our target language. With this strategy, we are able to use the operating system while developing it [14] as well as reuse existing device drivers.

There are a number of research projects aimed at designing an operating system with a strongly typed programming language with type inference, such as funk [1], snowflake-os [6], and house [2]. These projects have not been used as practical desktop or server operating systems, in place of Linux or BSD, however. The reason why is threefold.

All of these operating systems handle hardware interrupts by polling. Since the languages in which they are developed in are not reentrant, they must let the language runtime receive and store the hardware interrupts, forcing the operating system to poll the stored interrupts and raise events. On the contrary, practical operating systems receive hardware interrupts and call event handlers directly. To implement a polling operating system, one must therefore design from scratch, ignoring decades of research.

Due to this design difference, polling operating systems are incompatible with existing device drivers, which are written in C. The fundamental goal of an operating system is to run applications. To do so, it must detect the devices that are available and present an abstract interface to the applications. There are myriad devices in the world, and a polling operating system must reimplement the device drivers for many of them in order to be practical.

Achieving practical use is unlikely when the developers are busy reimplementing so many device drivers, but doing so is vital to the success of an operating system. Operating systems such as Linux are developed on systems that are running the operating system being developed, and the developers use the system for non-development purposes as well. This method of practical testing results in a constant improvement in the quality of the operating system.

For these reasons, we decided to implement a reentrant, strongly typed programming language compiler instead of using a poll-driven design.

Compiler Name	Binary Size	# Undef. Sym.	# Dep. Lib.
GHC-7.4.1	797228 B	144	9
SML#-1.2.0	813460 B	134	7
OCaml-4.00.1	183348 B	84	5
MLton-20100608	170061 B	71	5
jhc-0.8.0	21248 B	20	3

Table 1. Compiler benchmarks on simple programs that print "hoge" to standard output. We compare the size, number of undefined symbols, and number of library dependencies of the generated binaries.

3. Compiler Survey

We chose jhc [7] as the compiler for our research. We explain our decision process in this section.

3.1 Compiler Requirements for Snatch Design

Our goal is to snatch NetBSD into a strongly typed language with type inferencing. Many candidate languages exist, but the majority of them are not suitable for snatch design. In order to snatch a Unix-like kernel, the compiler must meet the following four requirements.

First, the compiler must be able to generate binaries that have few, if not zero, POSIX API calls. POSIX user-space applications can make use of rich libraries such as `libpthread` and `libc`, but such libraries are not accessible from kernel space. In order to run binaries in kernel space, one must replace any POSIX API calls with custom, error-prone C code. We minimize dependence on the POSIX API in order to avoid such code. An optimal compiler would generate binaries that require no libraries.

Second, the compiler must be able to generate executables that are as space and time-efficient as those generated by C. Kernel programs reside in memory and are executed frequently, so their efficiency is important.

Third, the compiler must generate binaries that are thread-safe and reentrant. The binaries must be thread-safe because kernels are event-driven and may run on multiple cores, where many instances of device driver calls are invoked in parallel. The reentrancy requirement is more complicated. Reentrancy is required in event-driven design, but it is also essential in implementing the interrupt handler that receives hardware interrupts, particularly when using a strongly typed language. Unix-like kernels have complicated interrupt handlers with many built-in algorithms. Our goal is to prevent errors in such interrupt handlers by using types. We need reentrancy in order to implement this.

Fourth, we must avoid global locking during garbage collection. Unix-like kernels are event-driven and often run on multicore CPUs, but global locking defeats the purpose of a multicore operating system. We also do not want global locking to impair interrupt response performance. It is considered impossible to completely remove global locking, but short locks are acceptable. The NetBSD kernel has interrupts every 10ms, so global locks should last under 1ms.

3.2 Compiler Benchmark

We considered five well-known, open-source compilers of typed languages: GHC, SML#, OCaml, MLton, and jhc. To test the first requirement, we implemented a simple program in all of the languages and compared the generated executables (c.f. Table 1). Smaller counts indicate less dependence on POSIX libraries. According to Table 1, jhc has the best characteristics among the candidates.

Table 1 also shows that jhc is space-efficient. A comparison between jhc and GHC [3] reports that jhc has many bugs, but

it is as time-efficient as GHC. Jhc therefore satisfies the second requirement.

Jhc does not meet the third requirement, however. Binaries generated by jhc are not reentrant. They have only one execution context and do not support threads at all. We have therefore developed the Ajhc Haskell compiler,[1] which adds reentrancy and thread support to jhc. This was achieved by the invention and implementation of Context-Local Heaps (CLHs) in Ajhc (c.f. §4).

The fourth requirement is automatically satisfied as a result of using CLHs. With CLHs, normal kernel functions and interrupt handler functions run in separate contexts. Garbage collection in one context does not block execution in other contexts. Global locking is still required when allocating the heap or context information structures, but the lock intervals are short and have negligible effect on system performance.

4. Context-Local Heaps

4.1 jgc: jhc's Garbage Collector

A context is a set of information and resources required for a program to execute. In jhc, context is implemented as global variables and arguments passed through function calls. Jhc generates binaries with only one context, so they are neither thread-safe nor reentrant. This restriction is imposed by jgc, jhc's garbage collector.

The garbage collection (GC) root pointer is an important part of the jhc context. It points to a single garbage collector, and the Haskell function mutators use the pointer to update the GC heap. When the jhc mutator needs to construct a new instance in the GC heap, it pushes a pointer to the instance onto the GC root. jgc can determine which instances are alive by scanning the GC heap from the GC root. The jhc runtime never switches contexts, neither actively (as coroutines would) nor passively (as hardware interrupts would).

Jhc functions can be separated into two groups based on garbage collector accessibility: functions written in C that cannot access the garbage collector, and functions written in Haskell that need to. All Haskell functions must therefore receive the GC root pointer as their first argument. All other parts of the context are accessible from both C and Haskell functions.

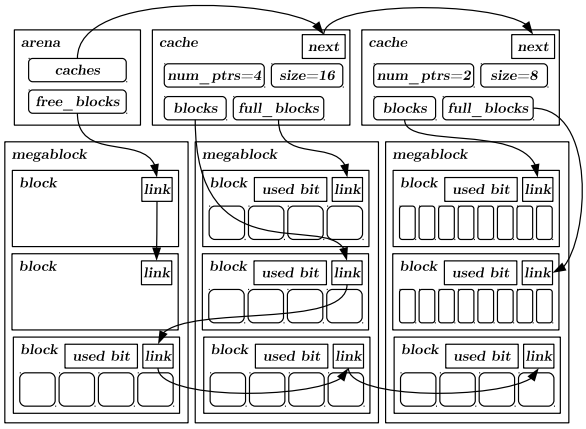

Figure 1. jgc Haskell heap internals

jgc manages the GC heap using megablocks and blocks (Figure 1). A megablock is the largest unit. One is allocated when the existing capacity of the GC heap is insufficient. Megablocks are segmented into blocks, which are segmented into chunks. jgc stores Haskell thunks into these chunks. Megablocks and blocks have a fixed size, which is set as a power of two during jhc runtime compilation.[2] The default sizes are 1MiB ($= 2^{20}$) and 4KiB ($= 2^{12}$) respectively.

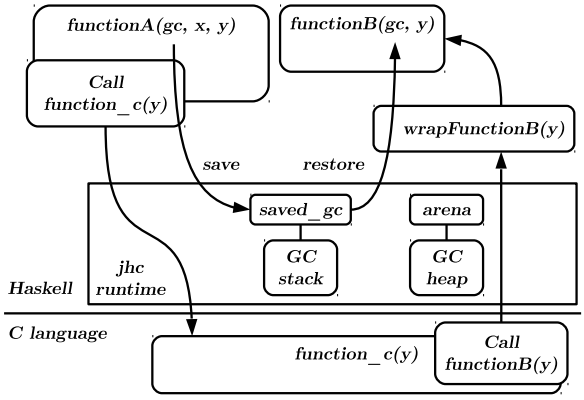

Figure 2. Interfacing with C using jhc

Using jhc, C and Haskell functions can call each other (Figure 2). A problem arises when a Haskell function (`functionA`) calls a C function (`function_c`) that in turn calls another Haskell function (`functionB`). `function_c` must pass the GC root pointer to `functionB` but does not have access to it! This problem is solved in jhc as follows.

All of the Haskell context except the GC root is accessed via a global variable named `arena`, which is initialized during jhc's runtime initialization and persists throughout the execution of the program. When a Haskell context calls a C function, the foreign function interface (FFI) mechanism saves the GC root pointer to a global variable named `saved_gc`. If the C function calls another Haskell function, the FFI retrieves the GC root pointer from that variable and passes it as the first argument to the Haskell context. In any further chains of Haskell function calls, the same GC root is passed as the first argument.

Due to this, there can only be one GC heap and one `arena` per program. In other words, there can be only one Haskell context, and multi-threading is not possible.

4.2 Support Multiple Contexts in Ajhc

Many Haskell implementations, including GHC, utilize a global heap (one GC heap per program). The global heap and purity of Haskell allow sharing of data between multiple contexts without having to copy it. It is difficult for one context to modify data inside the GC heap while another context is accessing the heap, however, making it difficult to implement a reentrant processing system.

In order to allow jgc to manage multiple Haskell contexts, Ajhc assigns a separate `arena` and GC heap to each Haskell context. We call these separate heaps *Context-Local Heaps* (CLHs).

Haskell contexts are not created during the initialization of the runtime. A new Haskell context is created when a Haskell function is called from C (Figure 3), and it is released when the function returns.

Each Haskell context consists of pointers to an `arena` and GC root. These pointers are passed as the first and second arguments of C functions within a Haskell context. They are allocated by NetBSD's kernel memory allocator, `kern_malloc()`. The Ajhc

[1] `http://ajhc.metasepi.org/`

[2] `http://ajhc.metasepi.org/manual.html#special-defines-to-set-cflags`

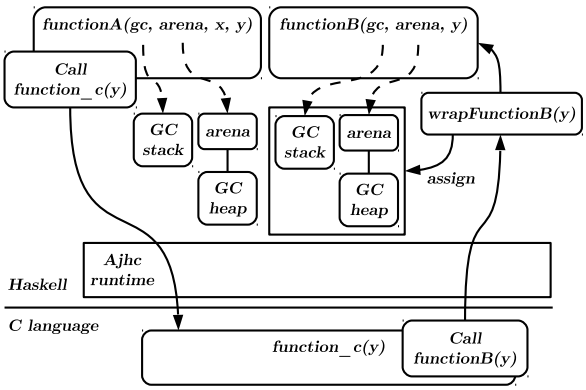

Figure 3. Interfacing with C using Ajhc

runtime caches the contexts internally instead of freeing them, in order to increase the performance of subsequent context generation.

Haskell constructors are called within a Haskell context. The Ajhc runtime attempts to ensure the memory of the instance by calling the `s_alloc()` function, finding and assigning free memory in the GC heap. A GC heap is not assigned to a context when it is created, and sometimes no memory in the GC heap is free. In such cases, the runtime assigns a new GC heap to the context by calling the `kern_malloc()` function. When the context is no longer needed, the GC heap is also cached internally for performance.

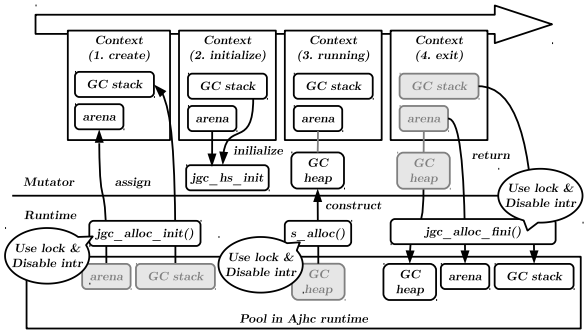

Figure 4. Life cycle of a CLH Haskell context

For the most part, a global lock is not required, but one is required when ensuring the `arena`, the GC root, and the GC heap, as well as returning them to the runtime. Since these structures are stored in the runtime cache, there is no need to call a memory allocation function, and the lock is generally completed in a short period of time. The global lock is implemented by the NetBSD `mutex(9)`,[3] which disables interrupts and spinlocks while holding the lock.

4.3 Context-Local Heap Pros and Cons

Use of CLHs has benefits as well as drawbacks. One benefit, due to reentrancy, is that it enables writing a hardware interrupt handler in Haskell, because sections are accessed exclusively by disabling interrupts using `mutex(9)`.

Another benefit is that garbage collection is done in parallel. A global lock is not held even while a context is performing garbage collection, so other contexts can continue to mutate data. The main context can receive hardware interrupts, and both the main context and interrupt context can be written in Haskell.

A third benefit is that the frequency of garbage collection is reduced in short-lived contexts. A clean GC heap is assigned at the beginning of a context, and the dirty GC heap that is returned to the runtime when the context is completed is reset to a clean state. When the capacity of the GC heap is sufficient, garbage collection is not performed at all. While garbage collection is of course performed on long-lived contexts (such as the page daemon of a virtual memory system[4]), event-driven programs, such as the NetBSD kernel that we are focusing on, tend to have short-lived contexts.

A drawback of using CLHs is that it becomes impossible to send and receive messages between contexts (via `MVar`). This disadvantage has not been significant in our rewriting of the NetBSD kernel, as a built-in tree/queue is used for passing messages within the kernel.

5. Snatching Drivers

We chose the AC'97 sound driver[5] as our initial snatch target. We reimplemented the member functions of `struct audio_hw_if` as the main context and the `auich_intr()` function as the interrupt context, all in Haskell (Figure 5).

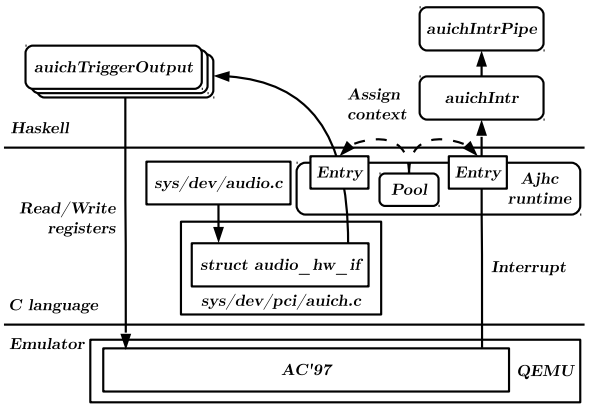

Figure 5. Partially snatched AC'97 sound driver

We initialize the Ajhc runtime during the NetBSD kernel initialization, just before autoconfiguration.[6] While this is too late to snatch NetBSD's core framework such as the virtual memory system, it is early enough to snatch normal device drivers.

Our modified kernel runs on QEMU and plays sound well. The interrupt context cannot be tested on QEMU, however, because QEMU does not strictly simulate AC'97 hardware interrupts. Unfortunately, we were unable to obtain AC'97 hardware.

We then snatched the HD Audio sound driver[7] in order to test the interrupt handler (Figure 6). Our modified kernel runs on real HD Audio hardware and successfully plays sound.

At this stage, the C and Haskell representations are almost identical, but we can refactor the Haskell code to use safer types later.

[3] `http://netbsd.gw.com/cgi-bin/man-cgi?mutex_enter`

[4] `http://netbsd.gw.com/cgi-bin/man-cgi?uvm`

[5] `http://netbsd.gw.com/cgi-bin/man-cgi?auich`

[6] `http://netbsd.gw.com/cgi-bin/man-cgi?config+9`

[7] `http://netbsd.gw.com/cgi-bin/man-cgi?hdaudio`

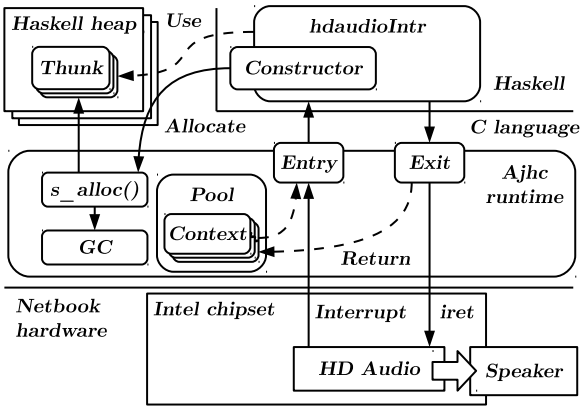

Figure 6. Partially snatched HD Audio sound driver

Kernel Version	Drivers	GC Option	Block Size
(O)	original		
(S)	snatched	default	4096 B
(N)	snatched	naive	4096 B
(B4)	snatched	default	16 B
(B5)	snatched	default	32 B
(B6)	snatched	default	64 B

Table 2. NetBSD kernel versions prepared for benchmarking. All kernel versions but (O) have AC'97 and HD Audio snatched via Ajhc. The (N) kernel uses naive GC (c.f. §6.1). The (B4), (B5), and (B6) kernels have particularly small GC blocks.

Kernel Version	Text	Data	BSS
(O)	11257927 B	564060 B	464540 B
(S)	11319534 B	565084 B	464540 B

Table 3. ELF section sizes.

6. Sound Driver Benchmarks

6.1 Benchmark Environment

How does our modified NetBSD kernel compare with the original kernel in terms of time and space efficiency? To benchmark the kernels, we used an environment as follows:

- Machine: Netbook / Acer Aspire One D260
- CPU: Intel Atom N455 @ 1.66GHz / 2 Cores
- Memory: 1GB
- HD Audio Codec: Realtek ALC272
- OS: NetBSD 6.1.2

We compared the GC performance of various kernels with different GC parameters (Table 2). Note that naive GC (N) maximizes GC frequency in order to maximize space efficiency.

6.2 Space Efficiency

We measured the space efficiency using both static and dynamic methods. First, we statically evaluated space efficiency by comparing the binary size of the kernel ELF (Table 3). The kernels with snatched sound drivers increased by 60KiB in the text section and 1KiB in the data section. This increase includes both the AC'97 and HD Audio drivers, and it is sufficiently small for practical use.

Kernel Version	Occupied Memory Size	CPU Load
(O)	46400 KiB	0.6%
(S)	48512 KiB	0.4%
(N)	48984 KiB	0.6%

Table 4. Occupied memory size and CPU load.

Kernel Version	# GC	Total	Average	Worst
(N)	7955	18.4 ms	0.0023 ms	0.0193 ms
(B4)		— kernel panic —		
(B5)	0	0 ms	0 ms	0 ms
(B6)	0	0 ms	0 ms	0 ms

Table 5. GC frequency and worst-case execution time.

Second, we dynamically evaluated space efficiency by reading the amount of free memory immediately after loading the kernel, using `/proc/meminfo`. The Haskell code is executed intermittently even when no sound is played because an interrupt to the HD Audio driver occurs at start-up. The kernels with snatched sound drivers take over 2MiB of memory more than the original kernel (Table 4), because some megablocks are held by jgc. Note that additional CPU cores or contexts would result in megablocks taking up more memory.

6.3 CPU Load

We measured time efficiency by getting the proportion of CPU load using the `top` command while playing the sound source (237 seconds, 44.1 kHz stereo[8]) with the `audioplay` command. Haskell code and garbage collection are not the dominant factor for CPU load, as the results (Table 4) show comparable CPU load among the various kernels.

6.4 GC Frequency and Worst-Case GC Time

We also measured worst-case execution time and frequency of garbage collection because time efficiency is not only measured in CPU load. For example, mutator throughput is decreased when GC suspends the context of hardware interrupt handlers many times. To measure these factors, we profiled Ajhc garbage collection in various kernels while playing the same sound source (Table 5).

Using naive GC (N) resulted in a worst-case execution time of 0.0193ms when under the 1ms lock limit discussed in §3.1. The worst-case execution time may be more significant, however, when snatching other parts of the NetBSD kernel that have more long-lived contexts. GC frequency was 33.5 times per second ($= 7995 \text{ times}/237 \text{ seconds}$) when using naive GC (N), with sound playing seamlessly.

With naive GC disabled, the GC block size is enough to run the event-driven mutator. Garbage collection is first performed in a context upon getting the 8th free block. 256 bytes ($= 8 \times 32$ bytes) is then enough to run the mutator in the (B5) case, which is quite small. In addition, the megablock size can be made smaller for even more space efficiency.

7. Related Works

The Rustic Operating System [5], written in the Rust programming language [4], has event-driven design. Rust has linear types and does not need garbage collection. Use of linear types is another good method of designing an event-driven operating system. The ATS language [13] also has linear types. In addition, both ATS

[8] `http://www.jamendo.com/en/track/771128/signal`

and Rust have a mechanism for using pointers more safely than in Haskell.

8. Conclusion and Discussion

We have developed Ajhc, a Haskell compiler that uses Context-Local Heaps to generate reentrant executables. We reimplemented some parts of the NetBSD kernel under the protection of the Haskell type system, using Ajhc. We demonstrated that we can implement hardware interrupt contexts as well as normal contexts in Haskell. As a result, we demonstrated the snatch design strategy–to gradually reimplement kernel source code in a language with type inferencing and garbage collection.

We have overcome the three challenges of making a functional operating system. The significant challenge of being able to use the operating system as we develop it is solved by utilizing the snatch design strategy. The hardware polling issue was avoided by implementing interrupt contexts in Haskell. The challenge of running alongside C device drivers was also eliminated by the snatch design strategy, allowing Haskell and C kernel components to work together without having to redesign the C components.

On the other hand, we have encountered some new problems. We describe three of them here.

One problem is the difficulty of debugging Haskell components of the kernel. Kernel programmers must deal with hardware issues as well as ordinary software bugs. When debugging hardware, binary traceability of source code is essential. Ajhc translates Haskell code into C code and then compiles the C code, however, and the semantics of the intermediate C code is very different from that of the source Haskell code. As a result, it can be difficult to trace Haskell code in the resulting binary. Kernel debuggers may prefer to use a functional language that is closer in representation to C than Haskell.

Another problem is that Ajhc's type system is not as powerful as GHC's. Ajhc does not support monad transformers,[9] to say nothing of various GHC extensions. We would like to call for readers who are familiar with type theory to join us in the development of Ajhc in order to solve this problem.

The third problem is that one cannot share values between distinct contexts in the current version of Ajhc. Deprived of the common Haskell heap, we are unable to implement `MVar`s [10] or `STM` [12]. We see distributed objects as a solution to this problem: connecting multiple contexts by network and implementing distributed garbage collection [8].

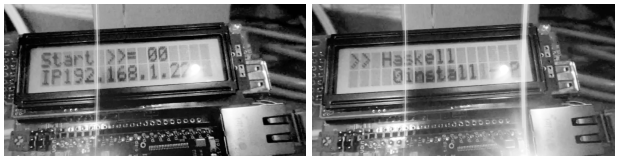

Figure 7. A Haskell RSS reader, compiled with Ajhc, runs using mbed-rtos on an ARM Cortex-M3 micro-controller with 512kB ROM and 32kB RAM. The left photo demonstrates getting the IP address with DHCP. The right photo shows a scrolling `reddit.com` news feed on the LCD panel. The `unsafeInterleaveIO` API connects the Ethernet interface, RSS parser, LEDs, and news text renderer on the LCD. See the demo video at `http://youtu.be/C9JsJXWyajQ`.

Despite difficulties, an Ajhc application on a tiny microcontroller tirelessly keeps telling the good news that the age of type-safe embedded systems development is coming soon (Figure 7).

A promising future awaits the redesign of a Unix-like kernel into Haskell. If we can assign types to the APIs of hardware, we can impose software invariants during the design of peripherals. Efficient filesystem designs will be enabled by the management of complex algorithms. Typed system call APIs will soon follow, allowing for safer and more productive applications programming. For example, typed APIs can enforce system call requirements such as memory alignments and buffer lengths. Moreover, type systems will be able to assert such invariants from system calls all the way to the hardware.

Such operating system design goals have been unattainably ambitious with C, and currently established operating systems such as Linux rely on human reviews for their safety. A strongly typed operating system, on the other hand, will gain a significant amount of safety from the type system. This will enable safe yet efficient embedded systems development that is important in the *Internet of Things* (IoT) where design of complex algorithms to run on cheap hardware is required.

Acknowledgments

This research is part of the Metasepi Project,[10] which aims to deliver a Unix-like operating system designed with strong types.

We thank Hiroki MIZUNO and Hidekazu SEGAWA for their assistance in the development of Ajhc. We thank Kazuhiro HAYAMA, Hiroshige HAYASHIZAKI, Keigo IMAI, Travis Cardwell, and Hongwei Xi for careful reading and commenting on the draft version of this paper.

We would also like to thank John Meacham for his ingenious jhc Haskell compiler. Finally, we would also like to thank the LPUX team for the experience and inspiration that led us to pursue the dream of a strongly typed operating system.

References

[1] T. funk team. Funk the functional kernel. URL `http://home.gna.org/funk/`.

[2] T. Hallgren et al. A principled approach to operating system construction in haskell. In *ICFP 2005*, 2005.

[3] D. Himmelstrup. Notes on the lhc: The great haskell compiler shootout. URL `http://lhc-compiler.blogspot.jp/2010/07/great-haskell-compiler-shootout.html`.

[4] G. Hoare. The rust programming language. URL `http://www.rust-lang.org/`.

[5] M. Iselin. Rustic operating system. URL `https://github.com/pcmattman/rustic`.

[6] jessica.l.hamilton. snowflake-os an o'caml operating system. URL `https://code.google.com/p/snowflake-os/`.

[7] J. Meacham. Jhc haskell compiler. URL `http://repetae.net/computer/jhc/`.

[8] D. Plainfossé and M. Shapiro. A survey of distributed garbage collection techniques, 1995.

[9] E. S. Raymond. *The cathedral and the bazaar*. O'Reilly, 2001. ISBN 9780596001315.

[10] S. F. SL Peyton Jones, A Gordon. Concurrent haskell, 1996.

[11] A. S. Tanenbaum and A. S. Woodhull. *Operating Systems Design and Implementation*. Pearson, 3 edition, 2008. ISBN 978-0-13-505376-8.

[12] S. P. J. Tim Harris, Simon Marlow and M. Herlihy. Composable memory transactions, 2005.

[13] H. Xi. Applied Type System (extended abstract). In *post-workshop Proceedings of TYPES 2003*, pages 394–408. Springer-Verlag LNCS 3085, 2004.

[14] G. P. Zachary. *Showstopper!* Free Press, 1994. ISBN 9780029356715.

[9] `https://github.com/ajhc/ajhc/issues/53`

[10] `http://metasepi.org/`