Systems Demonstration: Writing NetBSD Sound Drivers in Haskell

Kiwamu Okabe
METASEPI DESIGN
kiwamu@debian.or.jp

Takayuki Muranushi
RIKEN Advanced Institute for Computational Science
takayuki.muranushi@riken.jp

Abstract

Most strongly typed, functional programming languages are not equipped with a reentrant garbage collector. Therefore such languages are not used for operating systems programming, where the virtues of types are most desired. We propose the use of Context-Local Heaps (CLHs) to achieve reentrancy, which also increasing the speed of garbage collection. We have implemented CLHs in Ajhc, a Haskell compiler derived from jhc, rewritten 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.

1. Introduction

Reentrancy is necessary to achieve preemptive scheduling [3] 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 may seem easy 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 these circumstances.

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 interrupt 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. Since we do not have the manpower to write an entire operating system, we have adopted a rewrite design strategy, where we gradually rewrite components of an existing operating system in Haskell. In this paper, we invented and implemented Context-Local Heaps (CLHs, §2) to make jhc reentrant, and we call the result *Ajhc* [1]. Then we have rewritten 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.

ICFP'14, September 1–6, 2014, Gothenburg, Sweden
ACM 978-1-4503-2873-9/14/09.
http://dx.doi.org/10.1145/2633357.2633370

2. Context-Local Heaps

Many Haskell implementations 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 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, 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 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.

The mutator can run without any global locks. However the runtime requires a global lock at following cases: creating Haskell context, initializing context, allocating memory on GC heap and returning context 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)`, which disables interrupts and spinlocks while holding the lock.

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

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

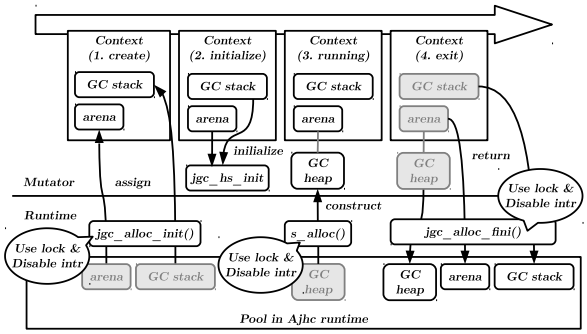

Figure 1. Life cycle of a CLH Haskell context

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), 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 an `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.

3. Rewriting Drivers

We rewrite the HD Audio sound driver in order to test the interrupt handler (Figure 2). 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.

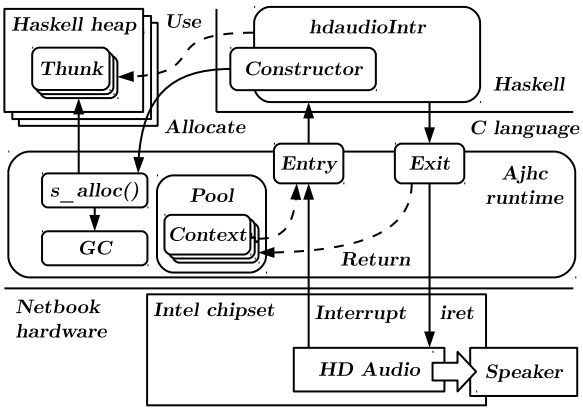

Figure 2. Partially rewrited HD Audio sound driver

4. Sound Driver Benchmarks

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

- Intel Atom N455 / 2 Cores / 1GB Memory / NetBSD 6.1.2

# GC	Total	Average	Worst
7955	18.4 ms	0.0023 ms	0.0193 ms

Table 1. GC frequency and worst-case execution time, with naive GC.

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

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) with the `audioplay` command. Haskell code and garbage collection are not the dominant factor for CPU load, as 0.5% among the various kernels.

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 1).

Using naive GC resulted in a worst-case execution time of 0.0193ms that is acceptable for Unix-like system. The worst-case execution time may be more significant, however, when rewriting other parts of the NetBSD kernel that have more long-lived contexts. GC frequency was 33.5 times per second when using naive GC, with sound playing seamlessly.

5. Related Works

The Rustic Operating System [2], written in the Rust programming language [1], has event-driven design. Rust has linear types and need no garbage collection. Use of linear types is another good method of designing an event-driven operating system. The ATS language [4] also has linear types. In addition, both ATS and Rust have a mechanism for using pointers more safely than in Haskell.

6. Conclusion

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 rewrite design strategy– to gradually reimplement kernel source code in a language with type inferencing and garbage collection.

Acknowledgments

This research is part of the Metasepi Project,[2] 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.

References

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

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

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

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

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