An Analysis of Haskell Parallel Programming Model in the HaLVM

Junseok Cheon[1], Yeoneo Kim[1], Taekwang Hur[1], Sugwoo Byun[2], Gyun Woo[1,3]

[1]Dep. of Electrical and Computer Engineering, Pusan National University
[2]Dep. of Computer Science, Kyungsung University
[3]Smart Control Center of LG Electronics

Email : [1]{jscheon, yeoneo, vhxpffltm, woogyun}@pusan.ac.kr, [2]swbyun@ks.ac.kr

Abstract. Recently, cloud computers are widely used due to the development of computing environment and network speed. Unikernel is considered to be an attractive operating system in the manycore environment that effectively provides cloud computers. HalVM is a unikernel coded in Haskell. In this paper, we analyze the performance of Haskell running in a unikernel, which has shown good parallel programming performance in a manycore environment. In this paper, we analyze the performance of Haskell running in a unikernel, while Haskell has shown good performance in a manycore environment. To do this, we select HaLVM as the target operating system and analyze the performance by testing various Haskell parallel programming models. Test result shows that HaLVM is scalable only on IVC. We also implement a library to make programming with IVC more convenient, and use this library to compare extensibility experiments. Our test shows HaLVM's scalability is 112.42% higher than Linux environment.

1. Introduction

Recently, due to the development of computing environment and high speed of network, many IT services are provided through network. Among them, cloud computers are running with environment over a network. Cloud computers are used not only for virtual server hosting in general use but also for games such as Google's dedicated cloud computer, Stadia.

In particular, as for effectively providing a cloud service for a dedicated purpose, an operating system, called a unikernel, has drawn attention. Unikernel is an application-specific operating system, also known as a library operating system [1]. Unlike conventional operating systems, unikernels are specialized for specific applications rather than for general purpose. They are also suitable for cloud environments. These unikernels are drawing attention not only in cloud services but also in manycore environments. On the other hand, Haskell supports good features of parallel programming.

This paper is to experiment the effectiveness of unikernel and parallel programming in Haskell on a manycore environment. To do this, we target HaLVM (Haskell lightweight virtual machine) [2,3], coded in Haskell, from a variety of existing unikernels. We implement various parallel programming models in HaLVM environment to analyze the program execution and performance.

This paper is organized as follows. Chapter 2 introduces unikernels as a related works. Chapter 3 introduces parallel programming models in Haskell to check their operations with HaLVM. Chapter 4 proposes an effective parallel programming model for HaLVM. In Chapter 5, we experiment with

HaLVM-based Haskell parallel programs and their scalability with existing Linux environments. Chapter 6 discusses the result of experimentation and, we conclude this work in Chapter 7.

2. Related work

This chapter introduces unikernel and HaLVM as a study on performance analysis of Haskell parallel programming based on HaLVM. Unikernel, unlike Linux, a general-purpose operating system, is an application-specific operating system, also called a library operating system. The basic concept of the unikernel is implement as a single user, a single address space, and a single process. This eliminates the need for kernel mode and user mode switching, so there is no overhead [4]. In addition, unlike the general-purpose operating system, unnecessary features removed. This shortens the time it takes to boot and improves security by reducing the area of attack.

These unikernels are research and implement in various ways, and the research directions are divided into two ways. First way is to support legacy code. This means that POSIX, which was used in Linux or UNIX, can be supported in unikernel so that existing programs can be used as is. Research achievements in this area include IncludeOS (C/C++), Osv (Java, Lia, Go), Rumprun (Ruby, Go, Python), Graphhene (C), and HermitCore (GCC).

The second way is to build clean-slate operating systems. This method began with the idea that the existing operating system is written in C, which makes it less secure and difficult to verify. Thus, the kernel is implemented from the beginning using a functional language that is more secure than an imperative language. Therefore, this operating system is not compatible with existing programs, but it has high safety features. These operating systems include MirageOS (Ocaml), HaLVM (Haskell), and LING (Erlang).

3. HaLVM

3.1. Architecture of HaLVM

This section describes the structure and features of HaLVM, the operating system used in this paper. HaLVM is an open-source unikernel developed by Galois with Haskell and was developed to solve the problem of microkernel-based operating systems. HaLVM is a unikernel based on Xen, a virtualization model, suitable for small, single-use, low-dependence programs. Many of the features of HaLVM are implemented in Haskell, as shown in Figure 1.

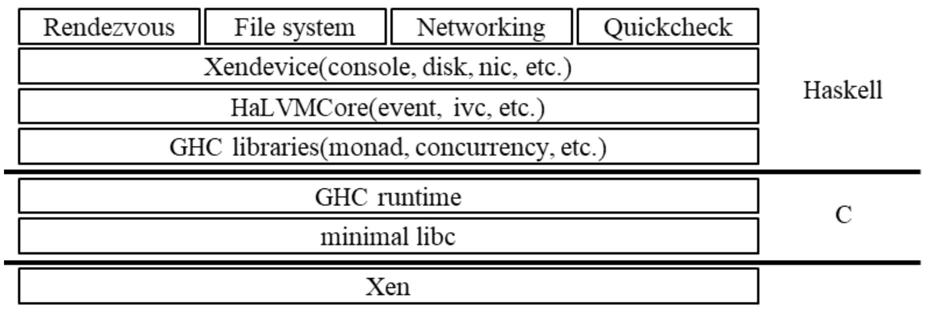
Rendezvous
File system
Networking
Quickcheck
Xendevice(console, disk, nic, etc.)
HaLVMCore(event, ivc, etc.)
GHC libraries(monad, concurrency, etc.)
Haskell
GHC runtime
minimal libc
C
Xen

Figure 1. Architecture of HaLVM

In Figure 1, the Rendezvous library is an important module for parallel programming. This module is provided for communication between unikernels in HaLVM. The communication using this module is referred to as IVC (inter virtual machine communication). IVC runs Rendezvous module in dom0 where hypervisor runs before program execution. Data using IVC is transferred to other virtual machines connected through Xen.

3.2. Haskell Parallel Programming Model in HaLVM
This section looks at the various Haskell parallel programming models running on HaLVM. Haskell offers a variety of parallel programming models, allowing you to choose a parallel programming model for your situation. However, this parallel programming model is based on Linux and does not necessarily work on HaLVM. We used five parallel programming models for operation verification.

The first parallel programming model is the Eval monad. The Eval monad is a model for executing parallel programs using parallel strategy [5]. The Eval monad performs parallel operations using spark, the smallest parallel unit provided by GHC. However, this model does not necessarily guarantee parallel execution.

The second parallel programming model is `forkIO` [6]. This model executes parallel programs using threads provided by the Haskell language. Because `forkIO` uses threads provided by Haskell, parallel units can be classified as green threads. In addition, Haskell threads are smaller than threads provided by the operating system, so they have a small overhead.

The third parallel programming model is `forkOS` [7]. This model runs parallel programs using threads provided by the operating system. In other words, the parallel unit of the model is a thread. And since `forkOS` is the same type as `forkIO`, the usage is the same. However, `forkOS` is characterized by running threads that use more memory than `forkIO`.

The fourth parallel programming model is Cloud Haskell [7]. This model based on the actor model [8]. In Cloud Haskell, message passing between nodes is via socket communication. The parallel unit of this model is a process So, the number of processes run is specified.

The fifth parallel programming model is IVC. IVC is a parallel model provided by HaLVM that uses Rendezvous mentioned in Section 1. The parallel unit of this model is a virtual machine. IVC differs from other models by exchanging data through programs installed in Xen dom0. Figure 1 summarizes the five parallel programming models and shows the results of execution verification on HaLVM.

Table 1. results of execution verification on HaLVM

Parallel Model	Unit of Parallelism	Running	Scalability
Eval monad	Spark	Yes	No
`forkIO`	green thread	Yes	No
`forkOS`	OS thread	No	-
Cloud Haskell	process	No	-
IVC	VM	Yes	Yes

Table 1 shows the analysis of Haskell parallel programming model in HaLVM. We wrote each program for Fibonacci numbers for each parallel programming model for analysis and ran it in HaLVM. We ran the program on a manycore environment and increased the number of CPU cores used by the program. At this time, if the execution speed is faster, the model is considered to be scalable. As shown in Table 1, there is no scalability in all parallel programming models except IVC. The IVC program is written in a message delivery style similar to Cloud Haskell, and has been shown to be scalable.

The cause analysis of the non-scalable parallel programming model confirmed that there was a communication problem between threads or processes. In the case of the Eval monad program, it was confirmed that spark is generated normally. However, there was no change in execution time after spark generation. forkIO similarly shows that tasks are created but no change in execution time. The `forkOS` had a problem that the program stopped after running. Finally, Cloud Haskell confirmed that an error occurs during program execution due to socket communication problems. This problem occurs because Cloud Haskell supports unicast communication, but HaLVM does not support unicast communication.

4. Develop a HaLVM based parallel programming model

This chapter introduces our proposed library for writing parallel programs effectively in HaLVM. We confirmed in Chapter 3 that HaLVM is only extensible in the IVC parallel model. However, IVC is a less convenient parallel programming method. This is because IVC must write VM programs to run in parallel and execute them separately. Therefore, this paper proposes an improved library as shown in Figure 2.

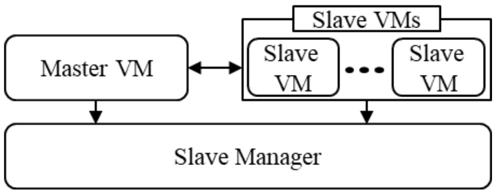

Figure 2. HaLVM based Parallel programming model

Figure 2 shows the parallel programming model we propose using IVC. This model consists of three modules. The first module is the Master VM. This module is a module in which functions other than functions to be parallelized operate. It then divides the data that needs parallelism into the desired number. There are two core tasks performed by the Master VM. The first is to request the slave node to run on the slave manager. The second is to connect the executed slave node with the IVC and transmit data.

The second module is the Slave Manager. This module runs Slave VMs and runs on Dom0 unlike other modules. This module is connected to Mater VM through IVC and executes slave by using shell script when slave execution request comes from Master VM. Also, unlike other modules, this module is written in C, not Haskell.

The third module is Slave VMs. This module processes the data received from the Master VM. Slave VM should be defined separately according to the program. Slave VM is connected to Master VM through IVC. After that, the data is received using the get function of IVC and calculated using a function in the Slave VM.

The proposed model has the advantage of being easier to use than the existing IVC. Existing IVC should write master VM program and slave VM program respectively and execute master VM and slave VM separately. However, the proposed model improves usability because the master and slave VMs are managed through Slave Manager.

5. Experiment

This chapter compares the results of running a program with IVC, a scalable parallel programming model, and the results of running a parallel program in an existing Linux environment. We used Haskell to write a program that calculates Fibonacci numbers for 39 ten times. The hardware environment used in the experiment was an Intel KNL 64-core single node, and the operating systems were Fedora 22 and HaLVM 2.4.0. The compiler used the same version of GHC 8.0.2. Figure 3 shows the results of measuring the scalability after running the program in both environments.

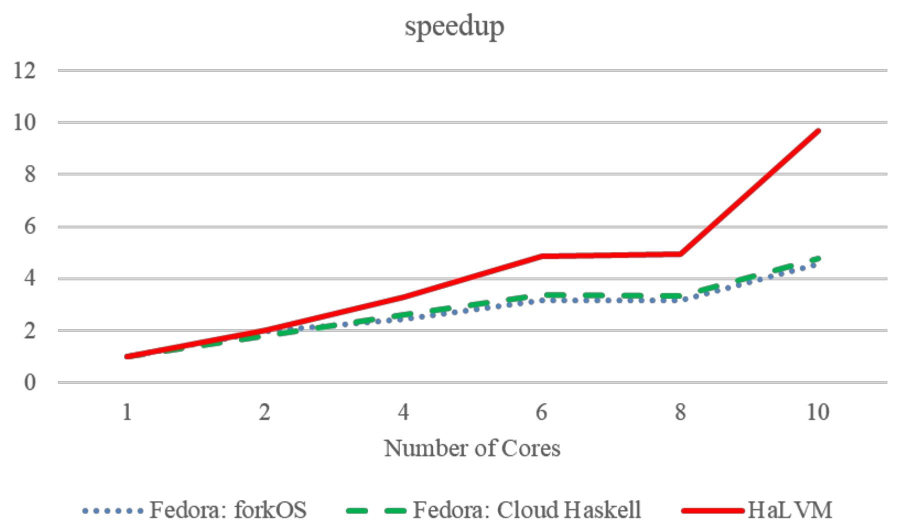

Figure 3. Comparison of Scalability

Figure 3 shows the result of comparing the scalability of parallel programs executed in Linux and HaLVM. In Linux, `forkOS` was used as a parallel model to test parallel performance as closely as HaLVM. HaLVM uses IVC as a parallel model.

The graph shows that HaLVM continues to grow as the number of cores increases. With two cores, you can see that HaLVM is 0.69% more scalable than Fedora environments. The scalability of 10 cores is 4.56 in Fedora `forkOS`. However, HaLVM measured 9.687, which is a difference of about 112.42%. This is because no context switch or kernel / user mode switch occurs when the number of cores increases. Therefore, we found that HaLVM is advantageous in the manycore environment with many cores.

6. Discussion

This chapter discusses why we have chosen scalability as an indicator for comparing performance between parallel programming models, and the issues of running four kinds of existing parallel programming models that did not run on HaLVM. The reason of choosing scalability is that the initial execution time of Linux environment and HaLVM environment are very different. A comparison of `forkOS` execution time in a single-core environment shows that it measured 28.8 seconds on Linux but 877.6 seconds on HaLVM.

This is a big difference, considering that HaLVM works in a virtual environment and in a Linux real environment. As a result, it is problematic to simply compare two operating systems based on execution time. This problem could be caused by Xen or HaLVM. Our goal was to check for scalability in a manycore environment when Haskell was applied to a unikernel. Therefore, we chose scalability as a comparative indicator.

On the other hand, most Haskell parallel programming models do not work well with HaLVM. Of the four parallel models in which the problem has been identified, the parallel programming model based on a per-thread needs to be checked for design intent. Unikernels are basically environments for a single process runs, but they also support multi-threads. However, as HaLVM is a newly created operating system, you may need to check the source code because thread may not have been provided or may be prohibited. In case of Cloud Haskell, as the problem is caused by communication, so it can be easily solved by changing the backend communication channel to IVC.

7. Conclusion

In this paper, we analyzed whether the performance of Haskell parallel programming can be secured by using unikernel in manycore environment. To do this, we chose HaLVM as the target unikernel and tested five parallel programming models. Test results show that parallel models other than IVC are not scalable in HalVM. However, finding that IVC also had a problem of pure usability, we have implemented a library to improve this issue. We also performed scalability test for parallel programming in Haskell to compare HaLVM and existing Linux kernels. The result showed that the scalability of HaLVM is superior to Linux as the number of cores increases.

In the future, we will explore ways to improve the four problematic parallel programming models in HaLVM as presented in Chapter 6. To this end, we will conduct a study comparing the RTS (runtime system) code of the existing GHC and HaLVM. In addition to implementing simple examples of system calls, we will analyze the performance between unikernel and Linux.

Acknowledgement

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT)(No. 2014-3-00035, Research on High Performance and Scalable Manycore Operating System)

References

[1] A. Madhavapeddy and D. J. Scott, "Unikernels: Rise of the virtual library operating system," Queue - Distributed Computing, Vol. 11, Issue 11, pp. 1-30, 2013.
[2] Adam Wick, "The HaLVM: A Simple Platform for Simple Platforms," XenSummit, 2012.
[3] acw, "The Haskell Lightweight Virtual Machine (HaLVM): GHC running on Xen," https://github.com/GaloisInc/HaLVM, (last checked 2018.12.03.).
[4] S. Cha, S. Jeon, Ramneek, J. Kim, Y, Jeong and S. Jung, "Trends in Unikernel and Its Application to Manycore Systems," Electronics and Telecommunications Trends, Vol. 33, No. 6, pp.129-138, 2018.
[5] S. Marlow, *Parallel and Concurrent Programming in Haskell: Techniques for Multicore and Multithreaded Programming*, 1st Ed., O'Reilly Media, 2013.
[6] S. P. Jones, A. Gordon and S. Finne, "Concurrent haskell," POPL, Vol. 97, pp. 295-308, 1996.
[7] S. Marlow, S. P. Jones and W. Thaller, "Extending the Haskell foreign function interface with concurrency," In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pp.22-32, 2004.
[8] J. Epstein, A. P. Black and S. Peyton-Jones,"Towards Haskell in the cloud," ACM SIGPLAN Notices, Vol. 46. No. 12. pp. 118-129, 2011.
[9] C. Hewitt, P. Bishop and R. Steiger, "A universal modular actor formalism for artificial intelligence," In Proceedings of the 3rd international joint conference on Artificial intelligence, pp. 235-245, 1973.