A Study of a Continuation-based Fine-grain Multithreaded Operating System CEFOS

Shigeru Kusakabe[1], Hideo Taniguchi[2], and Makoto Amamiya[3]

[1]Kyushu University,
Grad. Sch. of Info. Sci. & Elec. Eng.
Motooka, Nishi-ku Fukuoka
819-0395, Japan
kusakabe@csce.kyushu-u.ac.jp

[2]Okayama University
Grad. Sch. of Nat. Sci. & Tech.
Tsushima-naka, Okayama
700-8530, Japan

[3]Osaka Institute of Technology
Info. Sci. and Tech.
Kitayama, Hirakata, Osaka
573-0196, Japan

Abstract

CEFOS is an operating system based on a continuation-based zero-wait thread model derived from a data-flow computing model. A program consists of zero-wait threads, each of which runs to completion without suspension once started. Synchronization between zero-wait threads is autonomously performed in a dataflow manner according to their continuation relations. Handler routines for asynchronous events such as events from I/O devices are also realized with zero-wait threads and executed under the continuation-based multithreading mechanism. We can eliminate "interrupts" that interfere with the execution of instruction streams in typical conventional approaches. In this paper, we discuss implementation issues in CEFOS on FUCE. FUCE is a multi-core processor dedicated to the thread model and we can naturally handle concurrency and exploit parallelism in programs on FUCE even for I/O-centric computation. While we observe good scalability in terms of the number of execution units and I/O devices, relying solely on hardware in managing threads may expose its bottleneck. We also discuss and evaluate mechanisms for making up for a weak point of FUCE in handling I/O requests.

1 Introduction

Processors that exploit coarse-grain threads, such as SMT (Simultaneous Multi Threading) processors and CMP (Chip Multiprocessor), are widely available [3, 4, 10, 12, 13, 22]. Major operating systems, such as Linux, provide threads as a kind of process with reduced resources. We can write multithreaded applications in popular languages such as Java. However, as their basic execution model is a conventional sequential one, it seems difficult to deal with a considerably large amount of dynamically generated threads of various granularities that can be found in current or emerging software.

Multi-tasking operating systems not only handle concurrent tasks but also have concurrency and parallelism of various granularities in themselves. We claim that operating systems need to be developed based on a computation model that can deal with concurrency and parallelism of various granularities, while current major operating systems are developed based on the procedural and sequential computation concept. In order to investigate this claim, we are developing an operating system called CEFOS based on a continuation-based multithreading model derived from a dataflow computation model[1]. The concept of dataflow computation is suitable for processing parallel and concurrent activities of various granularities. There have been many research and development projects of dataflow architectures and related languages [5, 6, 7, 8, 9, 16, 18, 19, 20, 21]

A program for CEFOS consists of threads each of which runs to completion without suspension once started. We call a thread in our model a zero-wait thread in order to distinguish from other types of threads like Pthreads. Synchronization between zero-wait threads is performed in a dataflow manner according to continuation relations between them. A program can be seen as a dataflow graph in which a node corresponds to a zero-wait thread and an edge represents a continuation relation.

Handler routines for external events such as signals from I/O devices are also realized with zero-wait threads and executed under the continuation-based multithreading mechanism. We can eliminate "interrupts" that interfere with the

execution of instruction streams in typical conventional approaches, and we can naturally handle concurrency and exploit parallelism in programs even for I/O-centric computation. Execution control between zero-wait threads is performed autonomously according to continuation relations regardless whether threads are for computing some internal values, for managing shared resources, or for handling external events.

We implemented a prototype version of CEFOS by modifying Linux on commodity platforms and showed the effectiveness of our approach to some extent[23]. However, we also conduct a more model-oriented challenging approach, building our operating system from scratch on a dedicated platform. In this paper, we discuss mechanisms for handling I/O requests in our operating system on the FUCE (FUsion Communication and Execution) processor[2]. The FUCE processor is a continuation-based multithreading multi-processor dedicated to thread level parallelism.

Handling I/O requests is one of the potential bottlenecks in operating systems. In commodity operating systems, threads with an outstanding I/O request are set to a waiting state and listed in a wait queue associated with the event. In receiving the response, the current thread is interrupted and suspended, and threads in the wait queue are woken up to confirm the actual receiver of that response. Therefore, when the number of requests increases, the cost of such interrupts and activations considerably increases. On multiprocessor platforms, providing multiple I/O devices may improve the situation. However, distributing interrupts from the devices is one of the problems to achieve an ideal scalability.

Under the execution mechanism of CEFOS on FUCE, the receiver thread of the I/O response is explicitly specified without waking up other threads. The response from the device is converted into a continuation signal and the target thread is activated and executed in the same way as other threads without causing interrupt. This continuation-based direct activation mechanism of zero-wait threads enables us to exploit parallelism without complex interrupt management mechanisms.

CEFOS naturally exploits parallelism in operating system on a multithread machine like FUCE[11]. We can observe good scalability in terms of the number of execution units and I/O devices. However, naturally exploiting parallelism may lead to explosion of parallel activities which overwhelms hardware capacity while one of the important roles of operating system is hardware resource management. Basic execution mechanism is a dynamic dataflow architecture in which computation proceeds eagerly. The FUCE processor provides a self-continuation instruction which can be used in mutual exclusion. Relying solely on hardware in managing thread execution may result in a situation like livelock at the point where we introduce a gate to control eager execution for exclusive threads. The FUCE processor has only a non-priority queue, which can be overwhelmingly occupied by a number of threads issuing self-continuations. We discuss and evaluate our mechanism to make up for this weak point of FUCE in handling I/O requests. We address this issue by introducing software thread queues in front of the gate to control exclusive thread execution.

The organization of this paper is as follows. First, we explain the outline of the mechanism for handling multiple I/O requests in CEFOS on FUCE in Section 2. In Section 3, we evaluate the scalability of the I/O handling mechanism when changing the number of execution units and I/O devices. In Section 4, we examine our mechanism to prevent livelock situation due to increasing number of threads which check the availability of limited resources in a busy-wait manner.

2 Our Approach

In this section, we explain the outline of our handling mechanism for multiple I/O requests.

In handling external events from external devices, we must handle signals from the devices which are delivered asynchronously. In the typical interrupt-handling mechanisms for conventional platforms, these kinds of asynchronous events are treated as a kind of irregular events. Interrupt handling needs special mechanisms to satisfy the requirements to treat these kinds of irregular events, including resuming interrupted thread as early as possible, keeping the device busy for as short a time as possible, handling the nested interruption correctly, and so on.

In our model, handler routines for external events are also realized with zero-wait threads in the same way as other routines for internal computation and event handling is integrated into the continuation-based multithreaded execution mechanism. Figure 1 illustrates two different approaches in handling events from external devices. In our approach, we need not interrupt the running thread while we interrupt the running thread in the conventional interrupt-based approach. Simultaneous continuation events from different devices may activate different handler threads without nesting interrupt handlers. Threads are executed asynchronously, and a device which tries to activate the handler thread issues a continuation signal to the target thread then escape from their busy state quickly. Independent threads are executed in parallel and we can expect good scalability in terms of the number of execution units and devices. We can naturally handle concurrency and exploit parallelism which resides in programs even for I/O-centric computation.

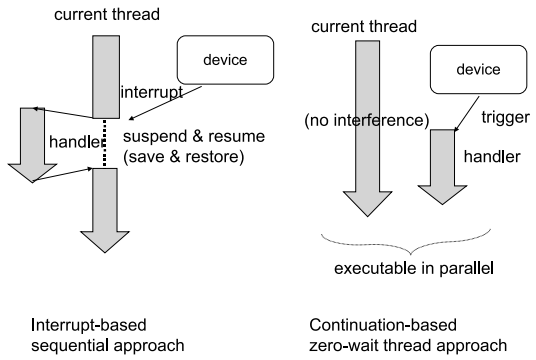

Figure 1. Two different approaches in handling external events from devices.

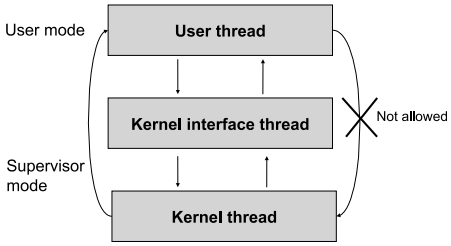

Figure 2. Mode changes between threads in the FUCE processor.

2.1 Execution control for shared resources

2.1.1 Execution mode change

Conventional processors have several execution modes represented as processor states and methods to change the modes. For instance, Intel x86 processors have `int 0x80` and `sysenter` instructions to realize mode change. Execution modes in the FUCE processor are set per thread. Each thread runs either in the user mode, the kernel mode or the kernel interface mode, and such threads are called as user thread, kernel thread, or kernel interface thread, respectively. Kernel threads run at the supervisor level and kernel interface threads support continuations from a user space to the kernel space.

Figure 2 shows possible continuations between different thread modes. In order to protect the kernel space, user threads cannot directly continue to kernel threads. To activate kernel threads from user threads as in system calls, user threads continue to kernel interface threads.

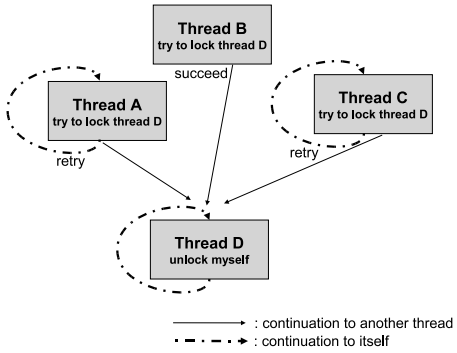

Figure 3. An example of mutual exclusion.

2.1.2 Mutual exclusion

Simultaneous invocations of the same code segment are allowed in our execution model in a way similar to those of dynamic-colored dataflow architectures. However, a mutual exclusion mechanism is necessary for critical regions which must not be executed simultaneously. There may be threads which are executed several times but which can not be executed simultaneously. For example, consider the case shown in Figure 3. In this example, three threads (Thread A, Thread B, and Thread C) try to continue to Thread D. However, only one thread can continue to Thread D at one time.

If we have a mechanism to lock a thread, we can achieve this mutual exclusion. Three threads try to lock Thread D. But, only one thread can lock thread D, and the thread which succeeded in locking Thread D (for example, Thread B) can continue to Thread D. Other threads which failed to lock Thread D continue to the starting point of themselves and try to lock thread D again. The critical thread, thread D in this case, also continues to the starting point of itself and resets the synchronization counter to a predefined value in order to wait for continuation signals from other threads which try to continue to Thread D.

2.2 Handling I/O requests with zero-wait threads

We explain the mechanism for handling I/O requests with zero-wait threads. The program structure is illustrated in Figure 4. The role of each thread is explained below.

`sender_thread:` System call requests from a user space must go through a `gate_thread` that is an interface to the kernel space. Threads issuing system calls try to lock the `gate_thread` at first (1-1). If a thread succeeds in locking the `gate_thread`, it continues to the `gate_thread` after delivering the necessary information such as the system call number, parameters to the

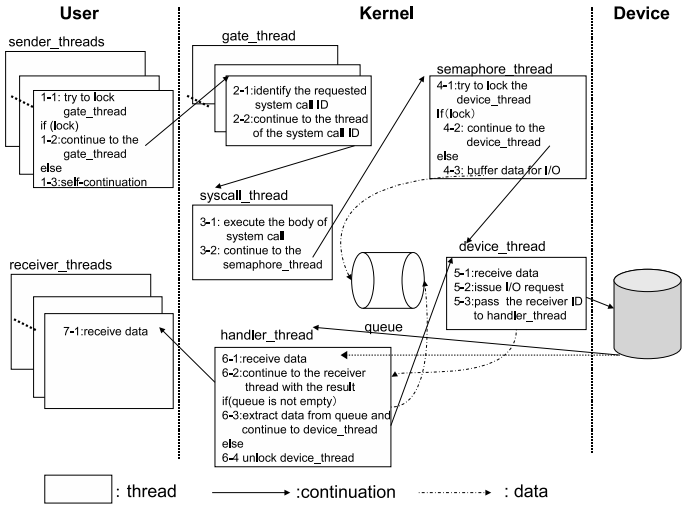

Figure 4. Handling multiple I/O requests with zero-wait threads.

system call and the receiver thread in the user space (1-2). Otherwise, it continues to the starting point of itself and tries to lock a `gate_thread` again (1-3). While `sender_threads` in user space are dynamically created in a dynamic dataflow mechanism, the number of `gate_threads` are statically fixed since they should be known to threads in user space as an entry point in a static manner. Each of the `gate_threads` cannot be activated simultaneously.

`gate_thread:` Threads in a user space cannot directly continue to the threads in the kernel space. This `gate_thread` works as an interface to the kernel space. This thread identifies the system call from the system call number and specifies the thread (`syscall_thread`) for the system call (2-1). Then, it continues to the thread after delivering the parameters to the system call and data to identify the `receiver_thread` (2-2).

`syscall_thread:` This thread is the system call body. In this case, as the system call is for an I/O operation, this thread continues to the `semaphore_thread` which guards the I/O device required by the I/O operation (3-1, 3-2).

`semaphore_thread:` First, this thread tries to lock the `device_ thread` which is a continuation of this thread (4-1). Then it continues to the `device_thread` if it can lock the thread (4-2). Otherwise, the I/O device is currently used and this thread enqueues data of the I/O operation in the queue before its termination(4-3).

`device_thread:` This receives data from the `semaphore_ thread` (5-1), and issues an I/O request to the device (5-2). Then, it continues to the `handler_thread` after delivering data to specify the `receiver_thread` (5-3).

`handler_thread:` The `handler_thread` is activated by the continuation signal from the device and receives the result data from the device (6-1). It executes a continuation instruction whose target is the `receiver_thread` and passes I/O data to the thread (6-2). Then it issues a continuation to the `device_thread` after delivering the data if there exist other I/O request data in the queue (6-3).

3 Scalability

In this section, we evaluate our handling mechanism for I/O requests on FUCE focusing on the scalability in terms of the number of thread execution units (TEU) and I/O devices.

As a typical conventional platform, consider the Linux Kernel 2.6 on a multiprocessing system. The Linux Kernel on a multiprocessing system has a run queue for ready threads (processes) per PE, and a wait queue for waiting threads per resource. In order to fully exploit the parallelism of the multiprocessing architecture, interrupts need to be delivered to any CPU in the system. Advanced Programmable Interrupt Controllers (APICs) such as the Intel APIC Architecture are designed as one of the attempts to efficiently deliver interrupts in multiprocessor computer systems. Interrupts requests in such systems can be distributed among the CPUs in two ways: static distribution or dynamic distribution. However, it is not easy to achieve an ideal scalability in handling I/O requests. We have to deliver the result to

the waiting thread in the multiprocessor system while taking care of load-balancing and mutual exclusion for critical resources. In our approach, the continuation signal directly activates the waiting thread without using a wait queue, and the activated thread is scheduled by means of the hardware property.

3.1 Preliminaries

In evaluation, we simulate the FUCE processor described in VHDL on the ModelSim simulator. The current specification of the FUCE processor is as follows: the size of Instruction Cache = 4KB/TEU, the size of Activation Control Memory = 40KB (5B/entry$\times$8 entries/page$\times$1K pages), the size of Thread Queue = 10KB (10B/entry$\times$1k entries), and size of Memory = 256MB. We use a program whose outline is shown in Figure 4. Programs of the operating system kernel and user applications issuing system calls are written in the FUCE assembly language. We assume the speed of FUCE processor is 1GHz. As we cannot connect real devices to our evaluation environment so far, we simulate a device as a `virtual_hw_thread` which occupies one execution unit during evaluation. This means that the `device_thread` continues to the `virtual_hw_thread` and the `virtual_hw_thread` continues to the `handler_thread` in Figure 4. This `virtual_hw_thread` has a loop and the RTT (Round Trip Time) of the simulated device is parameterized as the loop length. The maximum number of execution units is four in this evaluation.

We set RTT of the `virtual_hw_thread` as 0, 2, 4, and 6 micro seconds. We considered a system with multiple NICs (Network Interface Cards) of 10Gb Ethernet with TOE (TCP/IP Offload Engine). The estimated shortest interval of data packets is about 1 to 2 micro seconds in case of the maximum packet size of 1.5KB on 10Gb Ethernet. This interval amounts to 1.0×10^3 to 2.0×10^3 clock cycles when the speed of the FUCE processor is 1GHz.

We measured, as the throughput value, the maximum number of system calls completed within the fixed period without loss or delay. The fixed period was 1.0×10^5 clock cycles. We examine whether the maximum number of system calls scales with the number of TEUs and I/O devices.

3.2 Evaluation results

We consider a parallel I/O system which has a set of fixed pairs of a device and a processing unit. A device is tightly coupled to the paired processing unit, and interrupts from the device are handled by the specific processing unit. In such a system, we can improve throughput by N times when we have N device-processing pairs in an ideal situation while it is difficult to expect such an ideal situation.

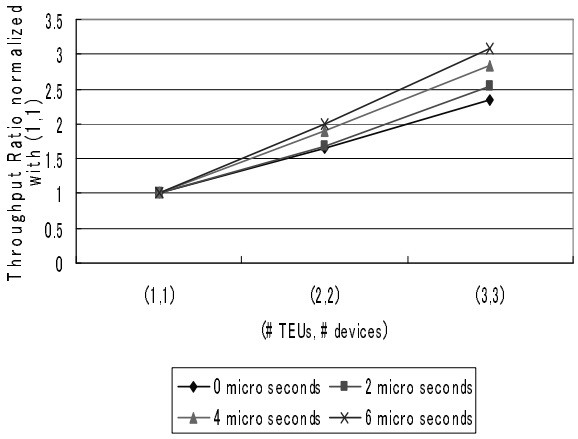

Figure 5. Relative throughput normalized compared to the case (the number of devices, the number of TEUs) is (1, 1).

As another assumption, we consider a system in which devices and processing units are loosely coupled. If we have a perfect load balancing mechanism for handling events from devices, we can also expect ideal scalability.

We examine whether our results are comparable to such ideal situations. Figure 5 shows the result. When RTT was 2 micro seconds, changing the pair of (the number of devices, the number of TEUs) from (1, 1) to (3, 3) improved throughput about 2.5 times. When RTT was 6 micro seconds, changing (the number of devices, the number of TEUs) from (1, 1) to (3, 3) improved throughput about 3.1 times.

Thus, we conclude we can expect good scalability in I/O handling in terms of the number of devices and the number of execution units. However, existence of a superliner case may mean a bad performance of the base case. One of the potential reasons is the problem of self-continuation in exclusive thread execution, we discuss in the next section.

4 Exclusive Thread Execution

Most operating systems are interrupt-driven systems which can provide low overhead and short latency at low interrupt frequency. However, they suffer from degradation situations at high interrupt frequency, such as receive livelock in which no useful work cannot be performed[14] as shown in Figure 6.

While CEFOS naturally exploits parallelism in operating system on a multithread machine like FUCE, naturally exploiting parallelism may lead to a similar situation due to explosion of parallel activities which overwhelms hardware capacity. In this section, we evaluate the mechanism

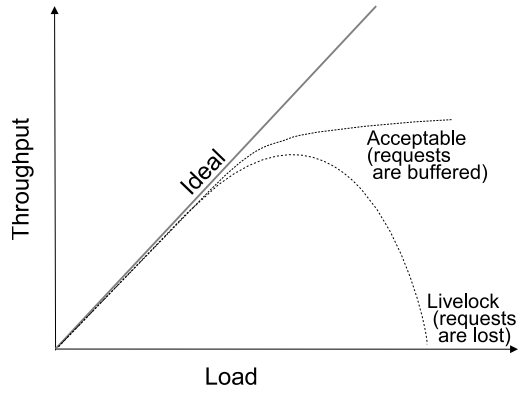

Figure 6. Overloaded systems may lead to receive livelock in which no useful work cannot be performed.

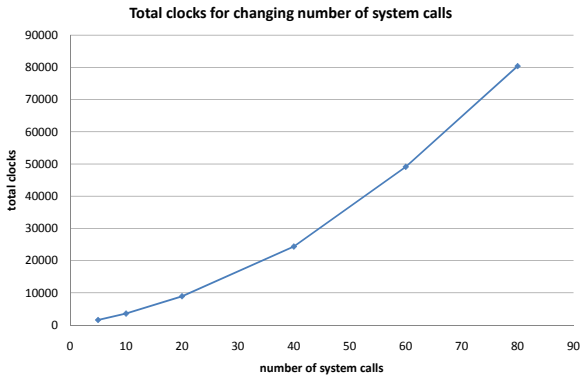

Figure 7. Total clocks for changing number of system calls.

to prevent a potential livelock situation due to increasing number of threads which check the availability of limited continuation threads in a busy-wait manner. We examine mechanisms to control contention for shared threads and resources as well as scalability. The FUCE processor provides a self-continuation instruction which can be used in mutual exclusion. Relying solely on hardware in managing thread execution may result in a situation like livelock at the point where we introduce a gate to control eager execution for exclusive threads. The FUCE processor has only a non-priority queue, which can be overwhelmingly occupied by a number of threads issuing self-continuations. We discuss and evaluate our mechanism to make up for this weak point of FUCE in handling I/O requests. We address this issue by introducing software thread queues in front of the gate to control exclusive thread execution.

4.1 Problem due to self-continuation

Figure 7 shows total clocks required to complete multiple system calls. We changed the number of system calls as 5, 10, 20, 40, 60, and 80.

The graph seems non-linear. If we increase the number of system calls n, the total clocks seem to grow in $O(n^2)$. We observed the clocks required per system calls increased linearly as shown in Figure 8. As the total clocks is the sum of clocks per system call, it grows in $O(n^2)$.

In order to find the cause of this situation, we analyzed simulation data. As shown in Figure 9, the number of self-continuations per `sender_thread` in user space has increased linearly. Although FUCE has a hardware thread queue, it is a non-priority queue. A self-continuation from a thread to reactivate the thread consumes one entry of the hardware queue. As the number of self-continuations increases, more queue entries are occupied without contributing the progress of actual computation.

4.2 Introducing software level thread queue

We introduce software level thread queues in order to avoid the situation in which the hardware level thread queue is occupied by the self-continuations checking the availability of limited resources in a busy-wait manner. In this case, we introduce software queues in front of `gate_threads`. A denied request for a `gate_thread` from a `sender_thread` is enqueued in software queue instead of issuing a self-continuation instruction, which will occupy an entry of the hardware queue. Figure 10 compares the total number of self-continuations when using software queues and those without software queues. We changed the number of system calls as the same as the previous subsection. As we can see from the figure, when introducing software queues, the increase of the self-continuations is almost proportional to the number of system calls without explosion like the case without software queues.

Figure 11 compares the total clocks required to complete system calls when using software queues and those without software queues. As we can see from the figure, when introducing software queues, the total clocks are significantly reduced as the number of system calls increases.

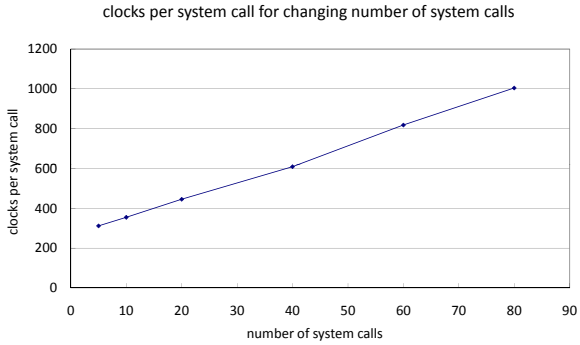

Figure 8. Clocks per system call in changing number of system calls.

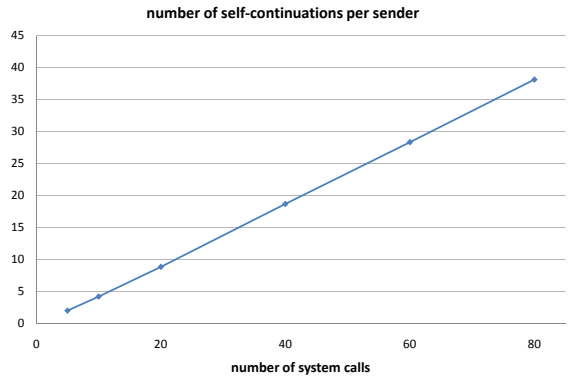

Figure 9. Number of self-continuations per sender thread in changing number of system calls.

5 Concluding Remarks

We discussed an operating system called CEFOS based on a dataflow based computation model. A program for CEFOS consists of zero-wait threads which run to completion without suspension once started. Synchronization among such threads is performed in a dataflow manner along continuation relations between threads. Handler routines for external devices are also realized with zero-wait threads and integrated in the continuation-based multithreaded execution mechanism. We can eliminate "interrupts" which disrupt the instruction stream inside the main processing unit in conventional platforms. In this paper, after introducing our model and our operating system based on the model, we discussed implementation issues on FUCE, which is a continuation-based multithreading processor dedicated to the thread level parallelism. We evaluated the scalability in throughput in terms of the number of execution units and I/O devices. We can naturally handle concurrency and exploit parallelism in programs even for I/O-centric computation. Self-continuations to realize mutual exclusion may cause a problem. Number of self-continuation threads may harmfully occupy the non-priority queue of FUCE processor in handling I/O requests. We addressed this issue by introducing software level thread queues and observed significant performance improvement for the increasing number of system calls.

References

[1] M. Amamiya. A New Parallel Graph Reduction Model and Its Machine Architecture. *Data Flow Computing, Theory and Practice*, pp. 445–464, 1991.

[2] S. Amamiya, T. Matsuzaki, M. Izumi, R. Hasegawa, and M. Amamiya. Fuce : The Continuation-Based Multithreading Processor. In *Proceedings of ACM International Conference on Computing Frontiers*, pp.213-224, 2007.

[3] J. Clabes, J. Friedrich, M. Sweet, J. DiLullo, S. Chu, D. Plass, J. Dawson, P. Muench, L. Powell, M. Floyd, B. Sinharoy, M. Lee, M. Goulet, J. Wagoner, N. Schwartz, S. Runyon, P. Restle G. Gorman, R. Kalla, J. McGill, and S. Dodson. Design and Implementation of the Power5 Microprocessor. In *Proceedings of DAC'04: the 41st annual conference on Design automation*, pp. 670–672, 2004.

[4] S. Gochman, A. Mendelson, A. Naveh, and E. Rotem. Introduction to Intel Core Duo Processor Architecture. *Intel Technology Journal*, 10(2):89–88, 2006.

[5] L. J. Hendren, X. Tang, Y. Zhu, G. R. Gao, X. Xue, H. Cai, and P. Ouellet. Compiling C for the EARTH Multithreaded Architecture. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, pp. 12–23, 1996.

[6] J. Hicks, D. Chiou, B. Seong Ang, and Arvind. Performance Studies of Id on the Monsoon Dataflow System. *Journal of Parallel and Distributed Computing*, 18(3):273–300, July 1993.

[7] H. J. Hum, O. Maquelin, K. B. Theobald, X. Tian, G. R. Gao, P. Cupryk, N. Elmasri, L. J. Hendren, A. Jimenez, S. Krishnan, A. Marquez, S. Merali, S. S. Nemawarkar, P. Panangaden, X. Xue, and Y. Zhu. A Design Study of the EARTH Multiprocessor. In *Proceedings of PACT '95: IFIP WG10.3 working conference on Parallel architectures and compilation techniques*, pp. 59–68, 1995.

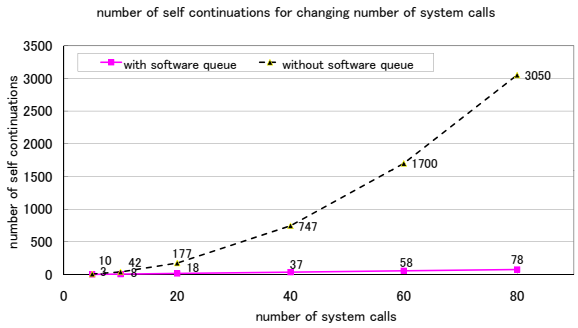

Figure 10. Comparison of number of self-continuations in changing number of system calls before/after using software queue

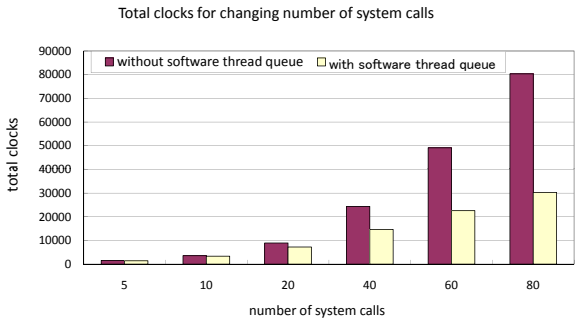

Figure 11. Comparison of total clocks in changing number of system calls before/after using software queue

[8] K. M. Kavi, H. Y. Youn, and A. R. Hurson. PL/PS: A Non-Blocking Multithreaded Architecture With Decoupled Memory And Pipelines. In *Proceedings of The Fifth International Conference on Advanced Computing (ADCOMP '97)*, 1997.

[9] T. Kawano, S. Kusakabe, R. Taniguchi, and M. Amamiya. Fine-Grain Multi-Thread Processor Architecture for Massively Parallel Processing. In *Proceedings of HPCA'95 (First IEEE Symposium on High-Performance Computer Architecture)*, pp. 308–317, 1995.

[10] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way Multithreaded Sparc Processor. *IEEE Micro*, 25(1):21–29, 2005.

[11] S. Kusakabe, M. Aono, M. Izumi, S. Amamiya, Y. Nomura, H. Taniguchi, and M. Amamiya. Scalability of Continuation-based Fine-grained Multithreading in Handling Multiple I/O Requests on Fuce Proceedeings of ACM International Conference on Computing Frontiers, pp.225–235, (5, 2007)

[12] J. L. Lo, S. J. Eggers, J. S. Emer, H. M. Levy, R. L. Stamm, and D. M. Tullsen. Converting Thread-Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading. *ACM Transactions on Computer Systems*, 15(3):322–354, 1997.

[13] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. Alan Miller, and Micheal Upton. Hyper-Threading Technology Architecture and Microarchitecture. *A hypertext history. Intel Technology Journal*, 6(1), 2002.

[14] J. C. Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-Driven Kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, 1997.

[15] W. A. Najjar, E. A. Lee, and G. R. Gao. Advances in the Dataflow Computational Model. *Parallel Computing*, 25:1907–1929, 1999.

[16] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A Multithreaded Massively Parallel Architecture. *SIGARCH Computer Architecture News*, 20(2):156–167, 1992.

[17] L. Roh and W.A. Najjar. Analysis of Communications and Overhead Reduction in Multithreading Execution. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, 1995.

[18] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An Architecture of a Dataflow Single Chip Processor. In *Proceedings of ISCA'89: the 16th annual international symposium on Computer architecture, New York, NY, USA*, pp. 46–53, 1989.

[19] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pp. 422–433, 2003.

[20] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. Wavescalar. In *Proceedings of the 36th annual IEEE/ACM International Symposium on MicroArchitecture*, pp. 291–302, 2003.

[21] B. Szymanski, editor. *Parallel Functional Languages and Compilers*. ACM Press Frontier Series, 1991.

[22] T. Ungerer, B. Robi.c, and J. Silc. A Survey of Processors with Explicit Multithreading. *ACM Computing Surveys*, 35(1):29–63, 2003.

[23] S. Yamada, S. Kusakabe, H. Taniguchi. Impact of Wrapped System Call Mechanism on Commodity Processors. In *Proceedings of ICSOFT 2006 (the 1st International Conference on Software and Data Technologies)*, pp. 308–315, 2006.