OS Mechanism for Continuation-based Fine-grained Threads on Dedicated and Commodity Processors

Shigeru Kusakabe[1], Satoshi Yamada[1], Mitsuhiro Aono[1], Masaaki Izumi[1],
Satoshi Amamiya[1], Yoshinari Nomura[2], Hideo Taniguchi[2], and Makoto Amamiya[1]

[1]Kyushu University,
Grad. Sch. of Information Sci. & Electrical Eng.
744 Motooka, Nishi-ku Fukuoka, Japan
kusakabe@csce.kyushu-u.ac.jp

[2]Okayama University
The Grad. School of Natural Sci. & Tech.
3-1-1 Tsushima-naka, Okayama, Japan

Abstract

Fine-grained multithreading based on a natural model, such as dataflow model, is promising in achieving high efficiency and high programming productivity. In this paper, we discuss operating system issues for fine-grained multithread programs. We are developing an operating system called CEFOS based on a dataflow based computation model. A program on CEFOS consists of zero-wait threads which run to completion without suspension once started. Firing control among such threads is performed in a dataflow manner along with continuation relations in the program. Target platforms include Fuce processor, which is dedicated to fine-grained multithreading, and commodity processors such as Intel x86. In this paper, after introducing our basic model and our operating system model, we discuss implementation issues on Fuce and commodity platforms. The evaluation results indicate that our approach on commodity platforms is effective in reducing overheads while our approach on a special architecture naturally exploit parallelism even in I/O handling.

1. Introduction

A number of applications are parallelized or multithreaded, and commodity processors that perform thread-level parallel processing, such as SMT (Simultaneous Multi Threading) processors and CMP (Chip Multiprocessor), are widely available[1, 2, 3, 4, 5, 6]. However, their underlying execution model is a sequential one, and process model of commodity operating systems assumes a sequential execution model. Writing or generating highly parallel multithread programs in imperative approaches seems a difficult problem even when the granularity of threads is large.

We believe fine-grained multithreading based on a natural model, such as dataflow model, is a promising approach. Fine-grained multithreading code can be hand-coded based on an appropriate model or generated from very high level languages such as functional or logic programming languages. Many architectures dedicated to such multithreaded programs have been proposed [7, 8, 9, 10, 11, 12, 13]. However, in this paper, we discuss issues on operating systems for fine-grained multithread programs.

We are developing an operating system called CEFOS based on a dataflow based computation model. A program for CEFOS consists of threads which run 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[1] Firing control among such threads is performed in a dataflow manner along with continuations. A program can be seen as a dataflow graph in which a node corresponds to a zero-wait thread and an edge corresponds to a continuation.

In this paper, we discuss implementation issues of operating systems for fine-grained multithreaded code on two different types of platforms. (See Figure 1) One is Fuce (FUsion Communication and Execution) processor architecture, which is dedicated to fine-grained multithreading[15], and the other is commodity processors such as Intel x86.

On Fuce, in addition to application programs, the operating system kernel is also multithreaded and we can

1-4244-0910-1/07/$20.00

[1]We will use the term "thread" and "zero-wait thread" interchangeably unless explicit explanation is necessary.

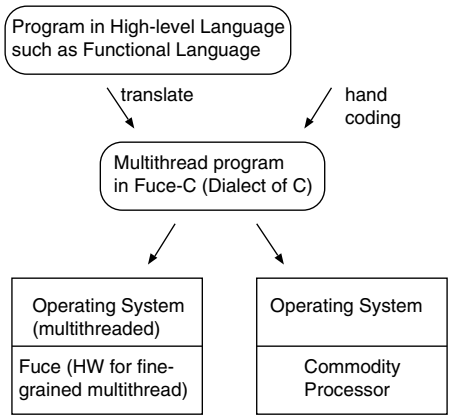

Figure 1. Our platforms.

naturally exploit parallelism in the operating system kernel. For example, interrupt handler routines consist of zero-wait threads and handling external devices is integrated in continuation-based multithreaded execution. Execution control between zero-wait threads are performed autonomously according to continuation relations regardless whether threads are for internal computation or for external event handling. Continuation-based activation mechanism of zero-wait threads enables us to easily exploit parallelism in programs even for I/O-centric computation without complex interrupt management mechanisms on Fuce.

On commodity platforms, we focus on mechanisms for user-level zero-wait threads. We implement a prototype version of CEFOS by modifying Linux on commodity platforms. While coarse grain threads like Pthreads are managed in a conventional way, we use scheduling techniques for fine-grained threads in order to reduce overheads of context switches on commodity processors[14].

The organization of this paper is as follows. First, we explain our basic thread model in section 2. Next, we introduce our operating system model on Fuce and discuss implementation issues focusing on handling I/O requests in section 3. Then we discuss the implementation issues of commodity version in section 4.

2. Our Thread Model

In this section, we explain our thread model. Programs using our thread model can be hand-coded based on our thread model or generated from very high level languages such as functional or logic programming languages.

2.1. Zero-Wait thread

Zero-wait threads have the following characteristics.

- A zero-wait thread consists of a synchronization counter and an instruction sequence. Instructions in the sequence of a thread are executed sequentially and may contain zero or more continuation instructions.
- A program consisting of multiple zero-wait threads seems like a dataflow graph whose nodes are zero-wait threads and edges are continuation relations.
- A continuation instruction specifies the starting points of succeeding threads and their context. It decrements the synchronization counter of the target thread by one. When the counter becomes to zero, the thread becomes ready to run.
- Once started, a zero-wait thread runs to completion without suspension. Execution starts at the beginning of the instruction sequence of the thread. It never starts from the middle of the thread.

As a zero-wait thread runs to completion once started, it has no wait points in the middle of the body like conventional posix threads. When the result of a request is returned after long latency, we use split-phase style transactions that separate the thread that receives the result of the request from the thread that issues the request. By using split-phase style transaction, latency hiding is realized naturally if enough number of ready threads exist.

2.2. Execution control of zero-wait thread

We discuss the basic mechanisms to execute zero-wait threads.

2.2.1 Context for threads

Contexts of threads are managed by using instance frames. Parallel and concurrent activities using the same thread code are activated with different instance frame like dynamic colored dataflow architectures. For example, an instance frame is created in a function invocation, and threads to compute the result of function invocation are executed using the same function instance frame.

2.2.2 Synchronization

A zero-wait thread may be specified as a continuation from more than one preceding threads. A continuation signal delivered from one single preceding thread may not make the target thread ready. Each thread has a synchronization counter. The initial value of the counter is set to the number of the preceding threads in typical cases. When a continuation instruction executes, it decrements the synchronization counter of the target continuation thread by one. If the counter becomes zero, the target continuation thread becomes ready.

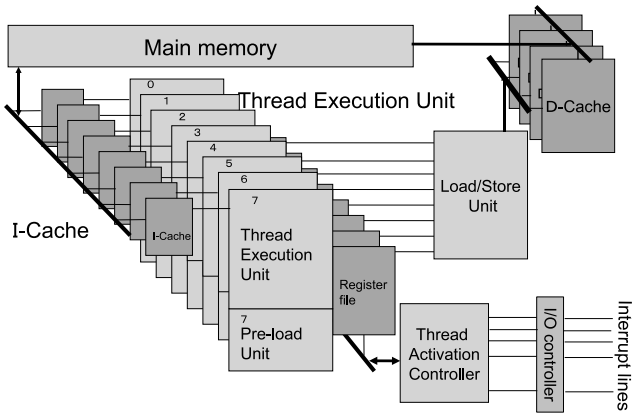

Figure 2. Overview of Fuce processor.

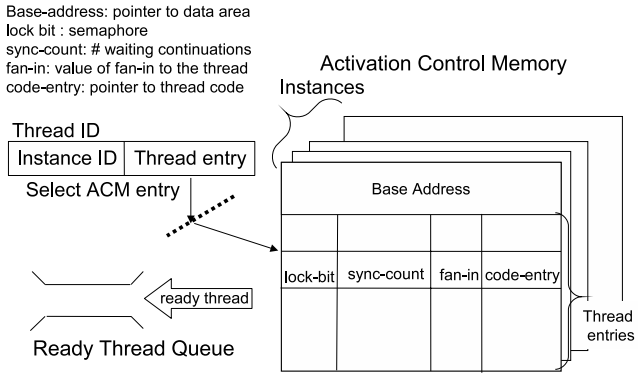

Figure 3. Components of TAC.

3. Operating System on Fuce

In this section, we introduce Fuce processor and our operating system model on Fuce. Fuce processor executes fine-grain non-preemptive threads and most features necessary to implement our operating system with zero-wait threads are provided at its hardware level. Since typical technical issues which are common in user applications are already discussed in another work[15], we discuss implementation issues of operating systems on Fuce focusing on handling I/O requests in this paper.

3.1. Fuce processor

Fuce processor is based on continuation-based multithreading, and threads of Fuce are almost the same as the zero-wait threads discussed previously. The thread model of Fuce is evolved from a dataflow computing model[10] although there are many types of thread in the literature[12, 16, 17, 18].

Figure 2 outlines an overview of Fuce processor. Multiple thread execution cores are implemented in a chip. The main components for thread execution on Fuce are Thread Activation Controller, Thread Execution Unit and multiple register files.

3.1.1 Thread Execution Unit

The Thread Execution Unit (TEU) executes instructions in a thread, and the number of TEUs in a chip is supposed as eight. The TEU consists of a Main unit and a Pre-loading unit. The Main unit is a very simple 32-bit RISC processor and its internal architecture is quite similar to the early MIPS processor. The instruction set of the Main unit is also derived from MIPS instruction set[19]. The Pre-loading unit is also a small RISC core, which executes instructions for data loading.

3.1.2 Thread Activation Controller

TAC controls all thread firings and mutual exclusions. Figure 3 illustrates the overview of the components of Thread Activation Controller (TAC). The TAC handles instructions for thread control issued in the TEUs, and updates the states of ACM explained below. A queue called ready-queue, is implemented inside the TAC. The TAC enqueues ready threads to the ready-queue. When a TEU finishes executing a thread and the corresponding register file becomes available, the TAC allocates a thread in the ready-queue to a free TEU.

Contexts for threads are managed in terms of instances. For example, an instance is created in a function invocation as a runtime environment for the function invoked, and threads to compute the result of the invocation are executed using the instance. Parallel and concurrent activities using the same thread code are activated with different instance contexts. Instance data is stored in a specially devised high speed-memory called Activation Control Memory (ACM). Each page in ACM is associated with an instance, and information of all the threads belonging to an instance is recorded in an ACM page. An ACM page consists of ACM-entries, each of which has sync-count, fan-in, code-entry and lock-bit to control the thread associated with the entry. Sync-count is the number of continuations for which a thread is currently waiting. Initial sync-count is set to the fan-in value. Code-entry is a pointer to the entry address of the thread code. Each thread is identified by Instance ID and Thread entry, its page number and page displacement in ACM. The lock-bit is used as a semaphore with the initial value of zero. The base-address is a pointer to the base address of data area used by an instance.

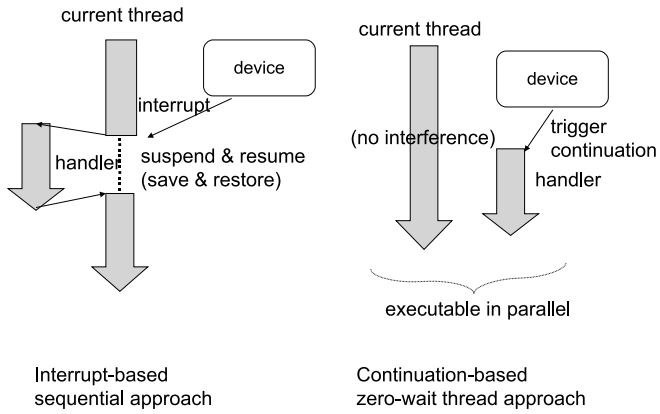

Figure 4. Two different approaches in handling an event from an external device.

3.2. Mechanism for operating system

3.2.1 Continuation from device

Figure 4 illustrates two different approaches in handling an event from an external device. Conventional approaches treat asynchronous events such as interrupts as a kind of irregular events. Interrupt handling needs special cares such as, resuming interrupted thread as early as possible, keeping device busy as short as possible, handling nested interrupts correctly, and so on.

In our operating system on Fuce, handler routines for external devices are also realized with zero-wait threads and event handling is integrated into the continuation-based multithreaded execution mechanism. We do not interrupt the current thread as in conventional approaches. Simultaneous continuation events from different devices may activate different handler threads without nesting handler threads. Devices send continuation signals and can 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 in programs even for I/O-centric computation.

An I/O controller of Fuce processor converts the signal from an external device into a continuation event and delivers the continuation event to the target handler threads. Signals from the same device are masked until the handler thread actually process the data from the device. The handler thread is activated through ACM in TAC and put into RTQ like other threads.

3.2.2 Execution mode change

Conventional processors have several execution modes as processor states. Execution mode in Fuce processor is set per thread. Each thread runs either in user mode, kernel mode or kernel interface mode, and is called as user thread, kernel thread, or kernel interface thread, respectively. Kernel threads run at supervisor level and kernel interface threads support continuations from user space to kernel space.

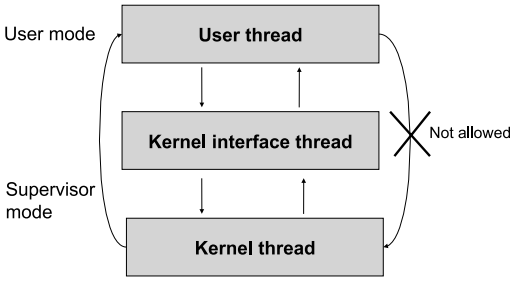

Figure 5. Mode changes between threads.

Figure 5 shows possible continuations between different thread modes. In order to protect kernel space, user threads cannot directly continue to kernel threads. To activate kernel threads from user threads as in system calls, user threads should continue to kernel interface threads.

3.2.3 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, mutual exclusion is necessary for critical regions, which must not be executed simultaneously. For example, consider the case where there is a thread which is executed several times but which can not be executed simultaneously as shown in Figure 6. In this example, three threads (Thread A, Thread B, and Thread C) try to continue to Thread D. But, only one thread can continue to Thread D at one time and other two threads must wait until Thread D becomes available again.

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 (for example, Thread B) can lock and continue to Thread D. Other threads which failed to lock Thread D continue to themselves in order to try to lock thread D again. The critical thread, thread D in this case, also performs self-continuation and resets the synchronization counter to a predefined value in order to wait for other continuation signals.

In Fuce processor, the following instructions are provided for mutual exclusion.

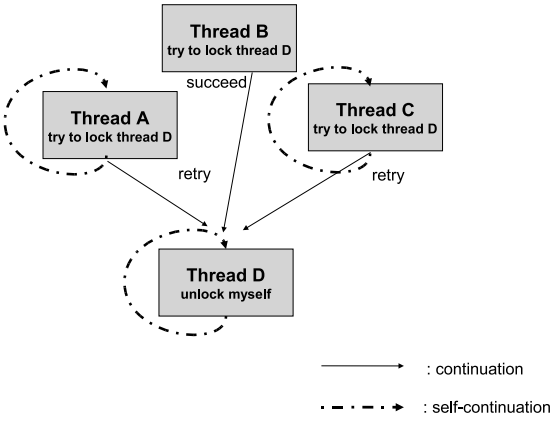

Figure 6. Mutual exclusion.

- `trylk Rd, thID` returns one and sets the lock-bit to one if the value of lock-bit of the target thread is zero. Otherwise, returns zero.
- `unlk thID` sets the lock-bit of the target thread to zero.

These instructions modify the lock-bit fields in ACM without accessing to the main memory. This is the different point from conventional test-and-set operations such as LL (load linked) and SC (store conditional) in MIPS architecture, which cause accesses to main memory. ACM can be accessed much faster than main memory.

3.3. Handling I/O requests with zero-wait threads

As a part of the operating system kernel on Fuce, we explain the mechanism for handling I/O requests with zero-wait threads. The interaction between threads in the mechanism is illustrated in Figure 7. The role of each thread is explained below.

sender_thread System call requests from user space must go through `gate_thread`, an interface thread to kernel space. Threads issuing system calls try to lock `gate_thread` at first (1-1). If it succeeds in locking `gate_thread`, it continues to `gate_thread` with necessary information such as system call number, parameters and the receiver thread in user space (1-2). Otherwise, it continues to itself and tries to lock `gate_thread` again (1-3).

gate_thread Threads in user space cannot directly continue to threads in kernel space. This `gate_thread` works as an interface to 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 parameters to the system call and data to identify `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 `semaphore_thread` guarding the I/O device required by the I/O operation (3-1, 3-2).

semaphore_thread First, this thread tries to lock `device_thread` which is a continuation of this thread (4-1). Then it continues to `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(4-3).

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

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

3.4. Evaluation

We evaluate our handling mechanism for I/O requests on Fuce focusing on the scalability in terms of the number of execution units and I/O devices.

3.4.1 Evaluation settings

We simulate Fuce processor described in VHDL on ModelSim simulator. The number of TEUs = 8, the size of Instruction Cache = 4KB/TEU, the size of ACM = 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. Throughput of TAC is 1 clock cycle and thread assignment from RTQ is 1 clock cycle.

Kernel program and user program issuing system calls are written in Fuce assembly language. We assume the speed of Fuce processor is 1GHz. As we cannot connect real devices to our evaluation environment, we simulate a device as a `virtual_hw_thread` which occupies one execution unit during evaluation. This `virtual_hw_thread` has a loop. RTT (Round Trip Time) of the simulated device is parameterized as the loop length. This means that `device_thread` in Figure 7 continues to `virtual_hw_thread` and `virtual_hw_thread` continues

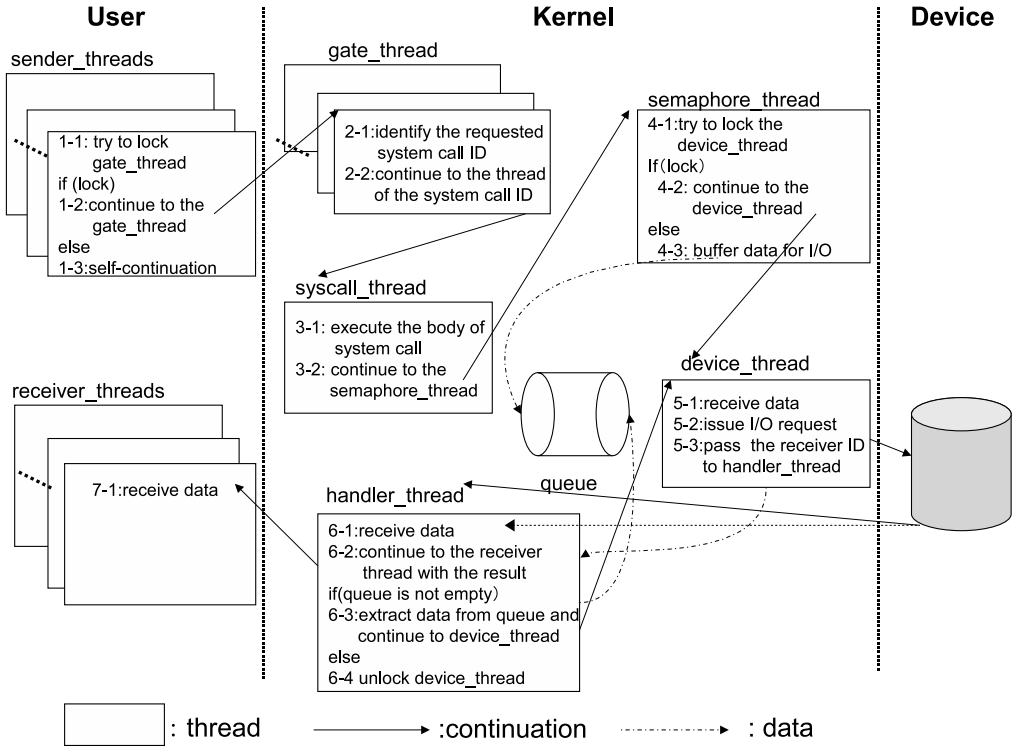

Figure 7. Handling multiple I/O requests with zero-wait threads

to `handler_thread` in Figure 7 after RTT. The maximum number of execution units which actually execute threads in benchmark programs is (the number of execution units in the processor) - (the number of devices) - 1, as we use one execution unit for managing the measurement.

We set RTT of `virtual_hw_thread` as 2 micro seconds. We considered a system with multiple NIC (Network Interface Card) of 10Gb Ethernet with TOE (TCP/IP Offload Engine). The shortest interval of data packets may be 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 cycles when the processor speed is 1GHz.

We measured the maximum number of system calls completed in the fixed period as throughput. We put N system call requests (`sender_thread`) into the execution units in the fixed period. The fixed period was 1.0×10^5 clock cycles. We gradually increased N, and measured the maximum number of system calls completed within the fixed period as throughput. When N exceeded the maximum number, requests for I/O exceeded the capacity of I/O handling mechanism, and outstanding system call requests were buffered in RTQ. We changed the number of TEU as 1, 2, 3, and 4, and the number of devices as 1, 2, and 3.

3.4.2 Evaluation results

Figure 8 shows the graph of throughput, where X-axis is the number of TEUs, Y-axis that of devices and Z-axis throughput. As we can see from the graph, increasing the number of TEUs improved throughput. In our system, events from devices are converted into continuations and threads triggered by the continuations are scheduled by RTQ at hardware level. We conclude handler threads and receiver threads were distributed over TEUs and executed efficiently.

If we consider a parallel I/O system which has a set of fixed pairs of a device and a CPU, where interrupts from a device are handled by the corresponding CPU. In such a system, we can improve throughput by N times with N device-CPU pairs in an ideal situation although it is difficult to expect such an ideal situation. We examine whether our results are comparable to such an ideal situation. Changing the pair of (*the number of devices*, *the number of TEUs*) from (1, 1) to (3, 3) improved throughput about 2.5 times. Thus, we can expect good scalability in I/O handling in terms of the number of devices and the number of execution units.

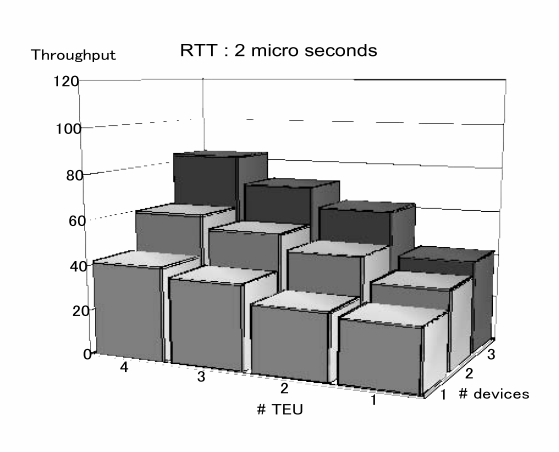

Figure 8. Throughput in handling I/O requests on Fuce

Table 1. Results of LMbench (Clock Cycles).

processor(GHz)	nullcall	2p/0K	2p/16K	L1$	L2$	Main
Celeron (0.5)	315	675	3235	3	11	93
Pentium4 (2.53)	1090	3298	5798	2	18	261
Core Duo (1.6)	464	1327	2820	3	14	152
PPC G4 (1.0)	200	788	2167	4	10	127

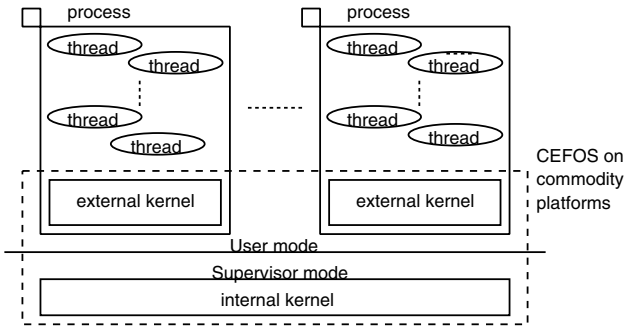

Figure 9. Overview of CEFOS on commodity platforms.

4. OS Mechanism on Commodity Platform

We try to show using fine-grained multithreading is also effective in achieving high throughput on commodity processors. CEFOS provides a split-phase style system call mechanism in which a request of a system call and the receipt of the system call result are separated in different threads. We can flexibly schedule threads in split-phase system calls in order to reduce overheads of system calls and enhance locality of reference.

While running user programs under the control of an operating system, frequent context switches and communications between user processes and the kernel are performed behind the scene. A system call requests a service of the kernel, and voluntarily causes mode-changes. Such activities involving operating system level operations are rather expensive.

Table 1 shows the result of a micro-benchmark LMbench [20] for platforms of commodity processors and Linux. The row "null call" shows the overhead of a system call and the row "2p/0K" shows that of a process switch when we have two processes of zero KB context. Thus, the row "*x* p/*y* K" shows the overhead of a process switch for the pair of *x* and *y* which represent the number and the size of processes, respectively. The rows "L1$", "L2$", and "Main" show the access latency for L1 cache, L2 cache and main memory, respectively. As seen from Table 1, activities involving operating system level operations such as system calls and context switches are rather expensive on commodity platforms.

Therefore, one of the key issues to improve system throughput is to reduce the frequency of context switches and communications between user processes and the kernel. In order to address this issue, we employ scheduling mechanisms based on a dataflow-like multithreading model in CEFOS.

Figure 9 shows the outline of the architecture of CEFOS on commodity processors: the external kernel in user mode and the internal kernel in supervisor mode. Internal kernel corresponds to the kernel of conventional operating systems. In the prototype CEFOS on commodity platforms, we use the kernel of conventional operating systems as the internal kernel.

A process in CEFOS has a scheduler for zero-wait threads and user processes schedule their ready zero-wait threads without involving the kernel to avoid causing overheads. Threads can be flexibly scheduled as long as dependencies among threads are not violated. The external-kernel mechanism in CEFOS intermediates interaction between the kernel and thread schedulers in user processes. In order to simplify control structures, process control or coarse grain thread control is only allowed at the switching points of zero-wait threads. Zero-wait threads in a process are not totally-ordered but partially-ordered, and we can introduce various scheduling mechanisms as long as the partial order relations among threads are not violated.

4.1. Display Requests and Data mechanism

Operating systems use system calls or upcalls [21] for interactions between user programs and operating system kernel. System calls issue the demands of user process through SVC and Trap instructions, and upcalls invoke spe-

cific functions of processes. The problem in these methods is overhead of context switches [22]. We employ Display Requests and Data (DRD) mechanisms [23] for cooperation between a user process and the kernel in CEFOS as we show below:

1. Each process and the kernel share a common memory area (CA).
2. Each process and the kernel display requests and necessary information on CA.
3. At some appropriate occasions, each process and the kernel check the requests and information displayed on CA, and change the control of its execution if necessary.

This DRD mechanism assists cooperation between processes and the kernel with small overhead. A sender or receiver of the request does not directly trigger the execution for the request at the instance the request is generated. If the sender triggers directly the execution at the receiver's side, the system may suffer from switching overhead. DRD mechanism assist to handles the request at our convenience with small overhead.

The external kernel mechanism in CEFOS intermediates interaction between the internal kernel and thread schedulers in user processes by using this DRD mechanism.

4.2. WSC mechanism

WSC mechanism buffers system call requests from user programs until the number of the requests satisfies some threshold and then transfers the control to the internal kernel with a bucket of the buffered system call requests. Each system call request consists of four items: type of the system call; arguments of the system call; the address where the system call stores its result; and ID of the zero-wait thread which should be activated after the system call.

The buffered system calls are executed like a single large system call and each result of the original system calls is returned to the appropriate thread in the user process. Figure 10 illustrates the control flow in WSC mechanism, and each number in Figure 10 corresponds to the explanation below.

1. A thread requests a system call to external kernel.
2. External kernel buffers the request of system call to in CA of DRD.
3. External kernel checks whether the number of requests has reached the threshold. If the number of requests is less than the threshold, the scheduler of zero-wait thread is invoked to select the next thread from the ready threads in the process. If the number of requests has reached the threshold, WSC mechanism sends the requests of system calls to the internal kernel to actually perform the system calls.
4. Internal kernel accepts the requests of system calls buffered in CA of DRD and executes them one by one.
5. Internal kernel stores the result of the system call to the specified address. Also, internal kernel tells the thread, whose ID is accepted as the fourth argument, that it stores the result. When internal kernel terminates executing all requests of system call, external kernel executes other threads. In other cases, WSC mechanism goes back to step 3 and repeats this transaction.

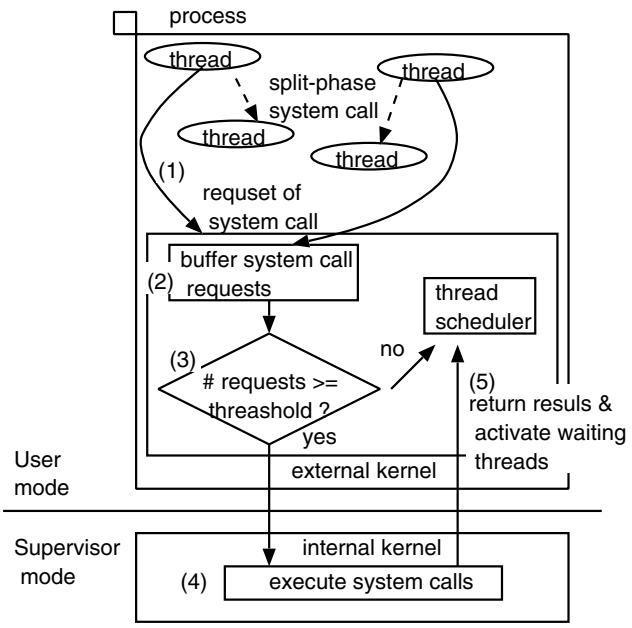

Figure 10. Control flow in WSC mechanism.

WSC mechanism reduces the number of mode changes between a user process and the kernel, which cause rather large overhead. Parameters and returned results of the buffered system calls under WSC mechanism are passed through CA of DRD to avoid frequent switches between the execution of user programs and that of the kernel.

4.3. Evaluation: reducing mode changes

In this subsection, we examine the effect in reducing overheads of system calls. We have modified Linux as the internal kernel and developed the external kernel on top of the internal kernel. CEFOS on commodity platforms can also accept system calls in the normal convention for the case buffering system call requests is not appropriate. We use only `getpid()`, which has a very thin body, as the target system call in order to clarify the system call overheads

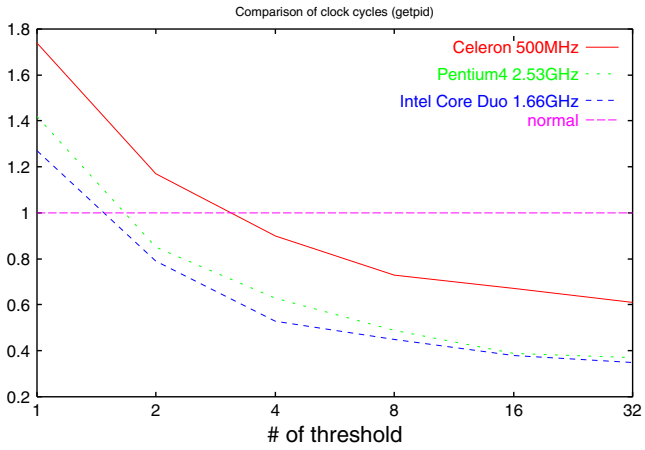

Figure 11. Relative clock cycles (`getpid`)

although WSC can handle heterogeneous system calls of various lengths.

We measured the number of clocks for a number of `getpid()` system calls using the hardware counter. We executed 128 `getpid()` system calls in our experiments. We changed the threshold of WSC as 1, 2, 4, 8, 16, and 32 for the WSC version. We also measured the total time of successive `getpid()` system calls under the normal system call convention in unchanged Linux.

Figure 11 shows the comparison results of clock cycles for `getpid()` system calls. The x-axis indicates the threshold of WSC and y-axis the ratio of clock cycles of WSC versions compared with clock cycles under the normal system call convention in unchanged Linux. The lower y-value indicates the better result of WSC.

As seen from Figure 11, we had additional overhead when WSC threshold is 1. However, we observe the effect of WSC when the threshold becomes 2 or larger for Pentium4 2.53GHz and Intel Core Duo 1.66GHz, and 4 for Celeron 500 MHz. The clock cycles in WSC versions are decreased as the threshold gets larger regardless of the processor type.

4.4. Evaluation : locality of reference

We examine the impact of WSC on exploiting locality of reference. We can expect high throughput when we can aggregate system calls among which we can enhance locality of reference by using WSC mechanism. We use chatroom benchmark[24] as a program for this evaluation.

The chatroom benchmark simulates chat rooms each of which includes both a server and user clients. The benchmark creates a number of processes each of which has `Sender` to send messages and `Receiver` to receive messages. The benchmark creates a number of TCP connections to send and receive a number of messages by using the same system calls. Each chat room has 20 clients by default, each client sends 20 messages whose size is 100B, and the server sends 19 messages in response to a single client message in our experiments. We apply WSC mechanism to `socketcall` system calls in the message sending part in the server of this program. The message sending part sends 19 messages per invocation, and is invoked 400 times during an execution.

The result in clock cycles for the WSC version is about 115.0 million cycles while about 133.0 million cycles for the normal version. The total number of clocks of the WSC version reduced to about 87% compared to that of the normal system call version. The reduction of about 18.0 million cycles by WSC mechanism is larger than the estimated reduction due to system call overheads, which is about 12.2 million cycles for 7,600 `socketcall` system calls. We consider this improvement is caused by enhanced locality of reference. Thus, we measured the number of events concerning memory hierarchies such as cache misses and TLB misses with a performance monitoring tool *hardmeter*[25].

The results shown in Table 2 were only measured for the message sending part which sends 19 messages due to the limitation of the event logging capacity. The threshold of the WSC version is 19, and the number of clocks of the WSC version reduced to about 80% compared to that of the normal system call version. We observed the events regarding memory penalties such as L2 cache misses and data TLB misses. As shown in Table 2, L2 cache misses of the WSC version reduced to 47% and data TLB misses to 92%. compared to that of the normal version. WSC mechanism for split-phase style system calls is also effective in exploiting locality of reference and reducing penalties concerning memory hierarchies such as cache misses and TLB misses.

Table 2. Results of chatroom benchmark

mechanism	# clocks	miss rate(%)	
		L2 cache	data TLB
normal	60216.7	1.01	2.78
WSC	48436.4	0.47	2.55
WSC/normal	0.80	0.47	0.92

5. Concluding Remarks

We discussed an operating system called CEFOS based on a dataflow based multithread model. A program for CEFOS consists of zero-wait threads which run to completion without suspension once started. Firing control among threads is performed in a dataflow-like manner along with continuations. On Fuce, the operating system kernel is also multithreaded as well as user applications. For example,

handler routines for I/O devices are also realized by using zero-wait threads and I/O handling is integrated in our continuation-based multithreaded execution. We can eliminate "interrupt" handling in conventional platforms. In this approach, we can naturally exploit concurrency in programs even for I/O-centric computation. We evaluated the scalability in throughput in terms of the number of execution units and I/O devices. We also discussed an approach in reducing overheads in commodity processors by using fine-grained multithreading. Our WSC mechanism in CEFOS buffers multiple system calls until some threshold is satisfied and then transfers the control to the operating system kernel with a bucket of the buffered system call requests. For chatroom benchmarks in which a number of threads issue system calls concurrently, the combination of our split-phase style system calls and WSC mechanism is effective in improving throughput by reducing mode changes and penalties concerning memory hierarchies such as L2 cache misses and TLB misses.

References

[1] 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, vol.15, no.3, pp.322-354, 1997.

[2] Masato Edahiro, Satoshi Matsushita, Masakazu Yamashina, and Naoki Nishi. A Single-Chip Multiprocessor for Smart Terminals. IEEE Micro, vol.20, No.4, pp.12-20, 2000.

[3] Deborah T.Marr, Frank Binns, David L.Hill, Glenn Hinton, David A.Koufaty, J.Alan Miller, and Micheal Upton. Hyper-threading technology architecture and microarchitecture: A hypertext history. Intel Technology Journal, Vol.6, No.1, 2002.

[4] 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, G. Gorman, P. Restle, R. Kalla, J. McGill, and S. Dodson, "Design and implementation of the POWER5 microprocessor," DAC '04: Proceedings of the 41st annual conference on Design automation, New York, NY, USA, pp.670-672, ACM Press, 2004.

[5] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: a 32-way multithreaded Sparc processor," Micro, IEEE, vol.25, no.1, pp.21-29, 2005.

[6] S. Gochman, A. Mendelson, A. Naveh, and E. Rotem, "Introduction to Intel Core Duo Processor Architecture," Intel Technology Journal, vol.10, no.2, pp.89-97, 2006.

[7] R.S. Nikhil, G.M. Papadopoulos, and Arvind, "*T: a multithreaded massively parallel architecture," SIGARCH Comput. Archit. News, vol.20, no.2, pp.156-167, 1992.

[8] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba, "An architecture of a dataflow single chip processor," ISCA '89: Proceedings of the 16th annual international symposium on Computer architecture, New York, NY, USA, pp.46-53, ACM Press, 1989.

[9] H.H.J. Hum, O. Maquelin, K.B. Theobald, X. Tian, X. Tang, 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," PACT '95: Proceedings of the IFIP WG10.3 working conference on Parallel architectures and compilation techniques, Manchester, UK, pp.59-68, 1995.

[10] Makoto Amamiya, "A New Parallel Graph Reduction Model and its Machine Architecture," Data Flow Computing, Theory and Practice Ablex Publishing Corp., pp.445-464, 1991.

[11] T. Kawano, S. Kusakabe, R. Taniguchi, and M. Amamiya, "Fine-grain Multi-thread Processor Architecture for Massively Parallel Processing," pp.308-317, IEEE Press, 1995.

[12] 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," Proceedings of the 30th Annual International Symposium on Computer Architecture, pp.422-433, 2003.

[13] Steven Swanson, Ken Michelson, Andrew Schwerin, MarkOskin. WaveScalar. Proceeding of the 36th annual IEEE/ACM International Symposium on MicroArchitecture, pp291-302, 2003

[14] Satoshi Yamada, et. al. Impact of Wrapped System Call Mechanism on Commodity Processors Proc. of ICSOFT 2006 (the 1st International Conference on Software and Data Technologies), Vol.1, pp.308-315, (2006,09)

[15] Takanori Matsuzaki, Satoshi Amamiya, Masaaki Izumi, and Makoto Amamiya. "A Multi-thread Processor Architecture Based on the Continuation Model. Proc of 8th Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA05), pp.83-90 (2005)

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

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

[18] T. Ungerer, B. Robi.c, and J. .Silc, "A survey of processors with explicit multithreading," ACM Computing Surveys, vol.35, no.1, pp.29-63, 2003.

[19] MIPS Technolocies, MIPS32 Architecture For Programmers Volume II: The MIPS32 Instruction Set.

[20] LMBench, http://www.bitmover.com/lm/lm-bench

[21] E. A. Thomas, et al "Scheduler Activation: Effective kernel Support for the User-Level Management of Parallelism, " Proc. of the 13th ACM Symp. on OS Principles, pp95-109, 1991.

[22] Purohit, A. and et al. "Cosy: Develop in user-land, run in kernel-mode." In *Proc. of HotOS IX: The 9th Workshop on Hot Topics in Operating Systems*, pages 109–114, 2003.

[23] H. Taniguchi, "DRD: New Connection Mechanism between Internal Kernel and External Kernel," Tran. of IEICE , Vol.J85-D-1, No2, 2002, in Japanese.

[24] Linux Benchmark Suite http://lbs.sourceforge.net/

[25] http://sourceforge.jp/projects/hardmeter