IMPACT OF WRAPPED SYSTEM CALL MECHANISM ON COMMODITY PROCESSORS

Satoshi Yamada and Shigeru Kusakabe
Grad. School of Information Sci. & Electrical Eng., Kyushu University
6-10-1 Hakozaki, Higashi-ku, Fukuoka, 812-8581 Japan

Hideo Taniguchi
Faculty of Engineering, Okayama University
3-1-1 Tsushima-naka, Okayama, 700-8530 Japan

Keywords: System call, mode change, locality of reference.

Abstract: Split-phase style transactions separate issuing a request and receiving the result of an operation in different threads. We apply this style to system call mechanism so that a system call is split into several threads in order to cut off the mode changes from system call execution inside the kernel. This style of system call mechanism improves throughput, and is also useful in enhancing locality of reference. In this paper, we call this mechanism as Wrapped System Call (WSC) mechanism, and we evaluate the effectiveness of WSC on commodity processors. WSC mechanism can be effective even on commodity platforms which do not have explicit multithread support. We evaluate WSC mechanism based on a performance evaluation model by using a simplified benchmark. We also apply WSC mechanism to variants of `cp` program to observe the effect on the enhancement of locality of reference. When we apply WSC mechanism to `cp` program, the combination of our split-phase style system calls and our scheduling mechanism is effective in improving throughput by reducing mode changes and exploiting locality of reference.

1 INTRODUCTION

Although recent commodity processors are built based on a procedural sequential computation model, we believe some dataflow-like multithreading models are effective not only in supporting non-sequential programming models but also in achieving high throughput even on commodity processors. Based on this assumption, we are developing a programming environment, which is based on a dataflow-like fine-grain multithreading model(Culler et al., 1993). Our work also includes a dataflow-like multithread programming language and an operating system, CEFOS(Communication and Execution Fusion OS)(Kusakabe and et al, 1999).

In our dataflow-like multithreading model, we use 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. Split-phase style transactions are useful in hiding latencies of unpredictably long operations in several situations. We apply this style to system calls and call as Wrapped System Call (WSC) mechanism. WSC mechanism is useful both in reducing overhead caused by system call mechanisms on commodity processors and in enhancing locality of reference.

In this paper, we evaluate the effectiveness of WSC mechanism on commodity processors. Section 2 introduces our operating system, CEFOS, and some of its features including WSC mechanism. Section 3 discusses the performance estimation and experimental benchmark results of WSC mechanism from the view point of system call overhead. Section 4 evaluates WSC mechanism for variants of `cp` program from the view point of locality of reference. We conclude WSC mechanism can reduce system call overhead and enhance locality of reference even on commodity platforms, which have no explicit support to dataflow-like multithreading.

2 SCHEDULING MECHANISMS IN CEFOS

2.1 CEFOS for Fine-Grained Multithreading

While running user programs under the control of an operating system like Unix, frequent context switches

Yamada S., Kusakabe S. and Taniguchi H. (2006).
IMPACT OF WRAPPED SYSTEM CALL MECHANISM ON COMMODITY PROCESSORS.
In *Proceedings of the First International Conference on Software and Data Technologies*, pages 308-315
DOI: 10.5220/0001321003080315

Table 1: Results of LMbench (Clock Cycles).

processor	null call	2p/0K	2p/16K	L1$	L2$	MainMem
Celeron 500MHz	315	675	3235	3	11	93
Pentium4 2.53 GHz	1090	3298	5798	2	18	261
Intel Core Duo 1.6GHz	464	1327	2820	3	14	152
PowerPC G4 1GHz	200	788	2167	4	10	127

and communications between user processes and the kernel are performed behind the scenes. A system call requests a service of the kernel, and then voluntarily causes mode change. Activities involving operating system level operations are rather expensive on commodity platforms.

Table 1 shows the result of a micro-benchmark LMbench (McVoy and Staelin, 1996) on platforms with 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 "MainMem" 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 mechanisms for efficient cooperation between the operating system kernel and user processes based on a dataflow-like multithreading model in CEFOS.

Figure 1 shows the outline of the architecture of CEFOS consisting of two layers: the external kernel in user mode and the internal kernel in supervisor mode. Internal kernel corresponds to the kernel of conventional operating systems. A process in CEFOS has a thread scheduler to schedule its ready threads.

A program in CEFOS consists of one or more partially ordered threads which may be fine-grained compared to conventional threads such as Pthreads. A thread in our system does not have a sleep state and we separate threads in a split-phase style at the points where we anticipate long latencies. Each thread is non-preemptive and runs to its completion without going through sleep states like Pthreads. While operations within a thread are executed based on a sequential model, threads can be flexibly scheduled as long as dependencies among threads are not violated.

A process in CEFOS has a thread scheduler and schedules its ready threads basically in the user-space. Since threads in CEFOS are a kind of user-level thread, we can control threads with small overhead. The external-kernel mechanism in CEFOS intermediates interaction between the kernel and thread schedulers in user processes. Although there exist some works on user level thread scheduling such as Capriccio (Behren and et al, 2003), our research differs in that we use fine-grain thread scheduling. In order to simplify control structures, process control is only allowed at the points of thread switching. 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. Thus, CEFOS has scheduling mechanisms such as WSC mechanisms and Semi-Preemption mechanism.

2.2 Display Requests and Data (DRD) Mechanism

Operating systems use system calls or upcalls (E.A.Thomas and et al, 1991) for interactions between user programs and operating system kernel. System calls issue the demands of user processes through SVC and Trap instructions, and upcalls invoke specific functions of processes. The problem in these methods is overhead of context switches (Purohit and et al, 2003). We employ Display Requests and Data (DRD) mechanisms (Taniguchi, 2002) for cooperation between user processes 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 of request at the instance the request is generated. If the sender triggers directly the execution of receiver's side, the system may suffer from

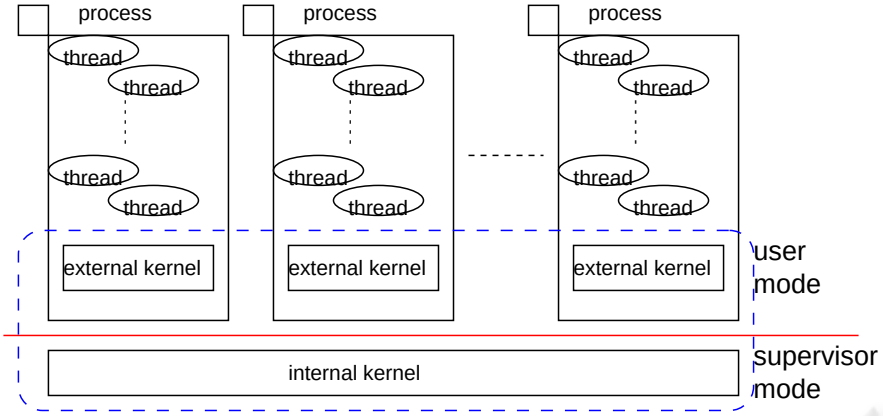

Figure 1: Overview of CEFOS.

large overhead to switch. On the other hand, the system handles the request at its convenience with small overhead if we use DRD mechanism. For an extreme example, all requests from a process to the kernel are buffered and the kernel is called only when the process exhausted its ready threads.

The external kernel mechanism in CEFOS intermediates interaction between the internal kernel and thread schedulers in user processes by using this DRD mechanism. Thus, CEFOS realizes scheduling mechanisms such as WSC mechanism and Semi-Preemption mechanism by using DRD mechanism.

2.3 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 kinds of elements listed below.

- type of the system call
- arguments of the system call
- the address where the system call stores its result
- ID of the thread which the system call syncs after the execution

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 2** illustrates the control flow in WSC mechanism, and each number in Figure 2 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 CA.
3. External Kernel checks whether the number of requests has reached the threshold. If the number of requests is less than the threshold, the thread scheduler is invoked to select the next thread from the ready threads in the process. If the number of request 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 and executes them one by one.
5. Internal Kernel stores the result of the system call to the address which Internal Kernel accepts as the third arguments of the system call. Also, Internal Kernel tells the thread, whose ID is accepted as the fourth argument, that it stores the result.
6. When Internal Kernel terminates executing all requests of system calls, External Kernel executes other threads. In other cases, WSC mechanism goes back to 3 and repeats this transaction.

WSC mechanism reduces overhead of system calls by decreasing the number of mode changes from user process to the kernel. 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.

3 EVALUATION: SYSTEM CALL OVERHEAD

We evaluate the effectiveness of WSC mechanism on commodity processors. The test platform is built by extending Linux 2.6.14 on commodity PCs.

Table 2: The values to calculate M (in clocks), and the estimated value of M.

processor (Hz)	T_{gen}	T_{sched}	T_{sync}	T_{req}	M
Celeron 500M	63	31	21	31	3.4
Pentium4 2.53G	110	43	27	31	1.24
Intel Core Duo 1.62G	61	30	19	29	1.42

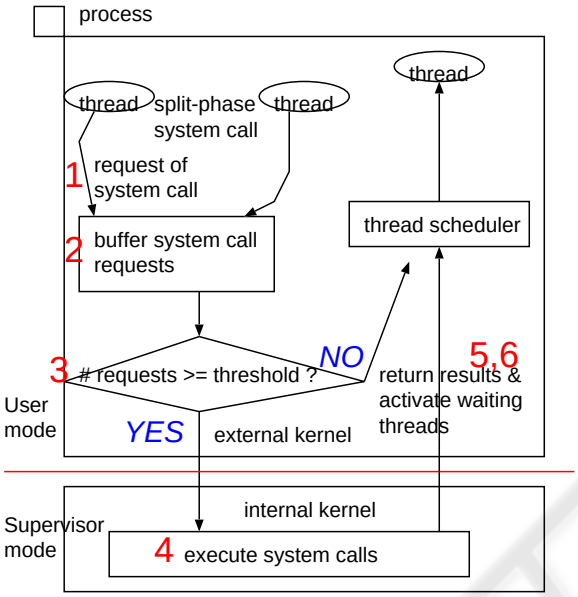

Figure 2: Control flow in WSC mechanism.

3.1 Estimation of the Effectiveness of WSC

First, we estimate the effectiveness of WSC mechanism by focusing on system call overhead. We compare the execution time of a program with normal system calls under the normal mechanism and that with split-phase system calls under WSC mechanism.

The total execution time of a program with N normal system calls under the normal mechanism, T_{nor}, is estimated as:

$$T_{nor} = T_{onor} + N \times (T_{sys} + T_{body}) + P_{nor} \quad (1)$$

where T_{onor} is the execution time of the program portion excluding system calls under the execution of the normal system call mechanism, T_{sys} is the setup and return cost of a single system call, and T_{body} is the execution time of the actual body of the system call. In this estimation, we assume that we use the same system call and that there exist no penalties concerning memory hierarchies such as cache miss penalties and TLB miss penalties in T_{onor} and T_{body}. P_{nor} is the total penalties including cache miss penalties and TLB miss penalties during the execution of the normal system call mechanism.

Programs to which we can apply WSC mechanism are multithreaded and use split-phase style system calls. Additional thread management should be performed in this multithreaded program and we describe the overhead of this additional part as T_{ek}. T_{ek} is estimated as:

$$T_{ek} = X \times T_{gen} + Y \times T_{sche} + Z \times T_{sync} \quad (2)$$

where X is the number of threads, T_{gen} is the overhead to generate a single thread, Y is the number of times threads are scheduled, T_{sche} is the overhead to schedule a thread, Z is the number of times synchronizations are tried and T_{sync} is the overhead of a synchronization.

Although the execution of system call bodies will be aggregated, buffering system call request must be performed for each system call. We represent the overhead of buffering a single system call request as T_{req}. Thus, T_{wsc}, the total execution time of a program with N split-phase system calls under WSC mechanism, is estimated as:

$$\begin{aligned} T_{wsc} =& T_{owsc} + T_{ek} + N \times T_{req} + \\ & [N/M] \times T_{sys} + N \times T_{body} + P_{wsc} \end{aligned} \quad (3)$$

where T_{owsc} is the execution time of the program portion excluding system calls, M is the number of system calls to be buffered for a single WSC (i.e. WSC threshold) and P_{wsc} is the total penalties concerning memory hierarchies including cache miss penalties and TLB miss penalties during the execution under WSC. We assume none of such penalties exists in T_{owsc} as in the estimation for T_{onor} and T_{body}.

ΔT, the difference between the execution time under the normal mechanism and that of under CEFOS with WSC is estimated as:

$$\begin{aligned} \Delta T =& T_{wsc} - T_{nor} \\ =& (T_{owsc} - T_{onor}) + \\ & \{T_{ek} + N \times T_{req} - (N - [N/M]) \times T_{sys}\} + \\ & (P_{wsc} - P_{nor}) \end{aligned} \quad (4)$$

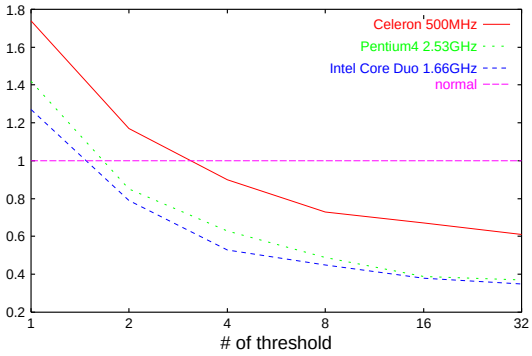

Figure 3: Comparison of clock cycles (`getpid`).

We can say the performance is improved by WSC mechanism when $\Delta T < 0$. We estimate the value of M, the number of system calls to be buffered, to satisfy this condition. We assume the following conditions for the sake of simplicity: each program portion excluding system calls is the same, and each system call body is the same both in the normal version and in CEFOS version. Under these assumptions, we will only observe the difference of system call cost between the normal version and the CEFOS version. This assumption makes $T_{owsc} - T_{onor}$ and $P_{wsc} - P_{nor}$ amount to zero. We also assume X, Y and Z are equal to N. Thus, we can estimate the condition for M to satisfy $\Delta T < 0$ as:

$$M > \frac{T_{sys}}{T_{sys} - (T_{gen} + T_{sche} + T_{sync} + T_{req})} \quad (5)$$

We measured each value in (5) in order to calculate the value of M that satisfies the above condition as shown in **Table 2** (we used the values of null call in Table 1 for T_{sys}) [1].

The performance on Pentium4 2.53GHz and Intel Core Duo 1.62GHz will be improved when M is larger 1. The performance on Celeron 500MHz will be improved when M is larger than 4. (Please note M is a natural number)

3.2 Performance Evaluation Using `getpid()`

The above estimation assumed each system call body is the same both in the normal version and in CEFOS version for the sake of simplicity. In this subsection, we examine our estimation by using `getpid()` as a system call to meet such an assumption. We measured

[1]We omit the values of PowerPC G4 because of the problem of accuracy. However, the observed M for PowerPC G4 is 4 according to the experiment explained in the next subsection.

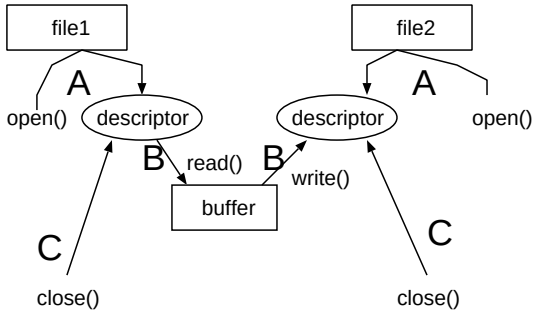

Figure 4: Control flow in cp program.

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 3 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 3, we have extra overhead when WSC threshold is 1, because of newly added load of T_{gen}, T_{sche}, T_{sync} and T_{req}. However, we observe the effect of WSC when the threshold becomes 2 for Pentium4 2.53GHz and Intel Core Duo 1.66GHz and 4 for Celeron 500 MHz as we estimated in the previous estimation. The clock cycles in WSC versions are decreased as the threshold gets larger regardless of the processor type.

4 EVALUATION: LOCALITY OF REFERENCE

In the previous section, we evaluate the effectiveness of WSC mechanism in reducing overhead caused by system calls. In this section, we examine the effectiveness in exploiting locality of reference. We can expect high throughput when we can aggregate system calls which refer to the same code or data. The test platform is also built by extending Linux 2.6.14 on Pentium4 2.53 GHz.

4.1 `cp` Program

We use modified `cp` programs to evaluate the effectiveness of WSC mechanism in exploit-

ing locality of reference. **Figure 4** shows an overview of the control flow in `cp` program, and the symbols A, B and C in Figure 4 correspond to the ones in the explanation below.

A. one `open()` system call opens a file to read and the other `open()` system call opens another file to write, preparing a file descriptor for each file respectively.

B. `read()` system call reads up to designated bytes from the file descriptor into buffer, and then `write()` system call writes up to designated bytes to the file referenced by the file descriptor from buffer.

C. `close()` system calls close these files.

Thus, a `cp` program uses six system calls per transaction. We use a `cp` program called NORMAL version, which executes these six system calls in the order we show above, like `open()`, `open()`, `read()`, `write()`, `close()` and `close()`. We have to `open()` a file before executing `read()` or `write()`, and we have to specify the file descriptor, which is the result of `open()` system call, to execute `read()`, `write()` and `close()`. Therefore, we cannot simply wrap these six system calls. We have to wrap two `open()` system calls and other four system calls respectively. Because of the additional overhead of using WSC mechanism that we mentioned in Figure 3, we cannot expect the effect when applying WSC mechanism to just one `cp` transaction. In fact, doing one `cp` in WSC version of one `cp` took about two times clock cycles compared to NORMAL version. Therefore, we consider doing multiple `cp`s in a program.

We use other four versions of `cp` program, and measure 11 portions of these 5 programs to observe:

I. whether WSC mechanism is effective or not in `cp` programs in total,

II. the difference between the effect of wrapping single type of system calls and that of wrapping various types of system calls, and

III. the effect of wrapping system calls which have the same code but refer to different data.

Figure 5 shows these 5 programs and 11 portions. "N" in Figure 5 is the number of `cp` transactions. Now, we explain each program and portion below. Then we explain why we choose these portions to examine the points of our interests above.

In Program 2 in Figure 5, we wrap every one of six kinds of system calls. We call this WSC+COLLECT version.

As a counterpart of this WSC+COLLECT, we also collect system calls of the same type in a block but execute the block with normal system call convention. We call this program as NORMAL+COLLECT version (Program 3 in Figure 5).

In addition, we implement WSC+RW and NORMAL+RW version (Program 4 and 5 in Figure 5), which change the order of `read()` and `write()` in WSC+COLLECT and NORMAL+COLLECT version.

Then, we show the explanation of 11 portions we measure.

1. from `open()` to `close()` of NORMAL.
2. from `open()` to `close()` of NORMAL+COLLECT.
3. from `open()` to `close()` of WSC+COLLECT.
4. from `open()` to `close()` of NORMAL+RW.
5. from `open()` to `close()` of WSC+RW.
6. `read()` and `write()` part of NORMAL+COLLECT.
7. `read()` and `write()` part of WSC+COLLECT.
8. `read()` and `write()` part of NORMAL+RW.
9. `read()` and `write()` part of WSC+RW.
10. only `write()` of NORMAL+COLLECT.
11. only `write()` of WSC+COLLECT

We measured only `write()` in portion 10 and 11 to observe the effect of wrapping system calls which refer to different data. While `read()` system call contains disk access time, `write()` system call buffers access to the disk and enables us to observe the effect of WSC mechanism excluding disk access time. Also, we implemented NORMAL+RW and WSC+RW and measured portion 6, 7, 8 and 9 to observe the effect of wrapping two types of system calls together. Then, we measured the whole `cp` in portion 1, 2 and 3 to examine if WSC mechanism is effective or not in total. Also, we measured portion 4 and 5 to examine the influence of wrapping `read()` and `write()` system calls on `cp` total.

We measure clock cycles and the number of events such as L1 cache misses in every portion. From these results, we investigate how WSC mechanism effects locality of reference from the view point of I, II and III above.

4.2 Performance Evaluation

Table 3 shows the result of `cp` programs. In this case, WSC threshold is 8 and we do `cp` transactions 100 times, which means N in Figure 5 is 100. The numbers in the row "portion" correspond to the numbers of the explanation we show in subsection 4.1. The row "#clocks" shows clock cycles, the row "L2$" shows L2 cache miss counts and rows "ITLB", and "DTLB" show the walk counts for ITLB and DTLB, respectively. We measured these events with a performance monitoring tool perfctr (Petterson, n.d.).

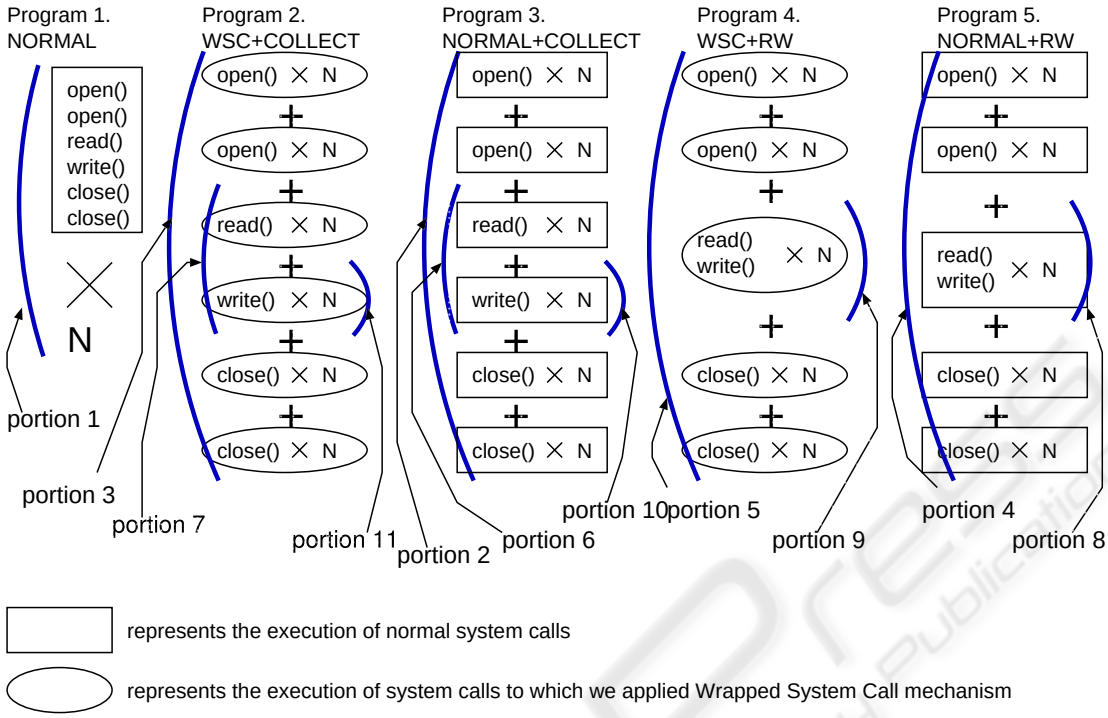

Figure 5: Program and Portion we measure in `cp` program.

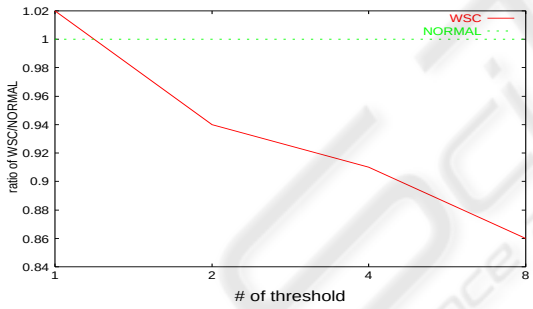

Figure 6: Comparison of clock cycles (`write()`).

In `write()` sections (portion 10 and 11), the clock cycles for WSC+COLLECT `write()` are less than NORMAL+COLLECT `write()` in about 0.12 million cycles, which is reduction to 83 % in clock cycles. The reduction of this 0.12 million cycles by WSC mechanism is larger than the reduction estimated by using formula in section 3, which is about 0.072 million cycles for 100 `write()` system calls. We consider this improvement is achieved by enhanced locality of reference, therefore we measure the number of events concerning memory hierarchies. As we expected, we can see the reduction of L2 cache misses, ITLB walks and DTLB walks in WSC+COLLECT `write()` compared to NORMAL+COLLECT `write()`. Thus, we can say WSC mechanism is effective even when each system call refer to different data. We changed the threshold and measured portion 10 and 11 to compare the results with those of `getpid()`. **Figure 6** shows the result, and we can see the same tendency as we see in Figure 3 that wrapping more than 2 system calls is effective in clock cycles in Pentium 4.

In read/write section (portion 6, 7, 8 and 9), we can see the effect of wrapping different system calls by comparing portion 6 with 7 and 8 with 9. Both clock cycles and number of events decrease in WSC version in both cases.

Finally, from portion 1, 2 and 3, we can say applying WSC mechanism to `cp` program is effective in total. When we compare portion 2 with 3 to ignore the difference of disk access pattern, the reduction in clock cycles is about 0.32 million cycles. As we see in `write()` system call, we can see the reduction of L2 cache misses, ITLB walks and DTLB walks. Therefore, we conclude that WSC mechanism for split-phase style system calls is effective in exploiting locality of reference. We can see the similar result from portion 4 with 5 and reach to the same conclusion.

Table 3: Results of `cp` program.

portion	#clocks	L2$	ITLB	DTLB
1. NOR	2,884,325	7378	511	136
2. NOR+COLLECT	2,588,200	8800	187	207
3. WSC+COLLECT	2,262,523	7740	81	120
4. NOR+RW	2,625,804	8264	227	200
5. WSC+RW	2,431,758	8118	128	197
6. NOR+COLLECT read/write	1,090,608	4385	112	69
7. WSC+COLLECT read/write	876,703	3503	37	42
8. NOR+RW read/write	1,096,045	3875	130	72
9. WSC+RW read/write	985,227	3647	70	62
10. NOR+COLLECT write	686,883	1779	93	28
11. WSC+COLLECT write	569,206	1363	41	13

5 CONCLUSION

In this paper, we discussed our WSC mechanism in CEFOS. While CEFOS is based on a dataflow-like fine-grain multithreading model, WSC mechanism is effective in improving throughput even on commodity platforms which have no explicit support to dataflow-like fine-grain multithreading.

Today, many investigation have been made about utilizing multithreading processor, such as SMT. Many of them tackle with memory hierarchy problem because cache conflict often occurs under the condition where several threads run concurrently. One effective solution to this problem is improving the scheduling of thread, which is conventional Pthread, to utilize CPU resources more effectively(Snavely and Tullsen, 2000). On the other hand, our work split conventional thread and control the thread in user process. Thus, we have more chances to schedule fine-grained threads more flexibly with smaller overhead.

In `cp` program, 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 walks.

Recently, the overhead of system call and context switch is increasing on commodity processors. Besides, we think the tendency continues that latency of memory access becomes bottleneck, which is coming from the gap between processor speed and memory speed. Therefore, we think WSC will be more effective in the future, which can reduce the overhead of system call and context switch and enhance the locality of reference. We believe this will contribute to higher throughput of internet server and large-scale computation in the future. Our future work includes collecting more data from other processors and exploiting the effect of SYSENTER/SYSEXIT command in x86 architecture.

REFERENCES

Behren, R. and et al (2003). Revising old friends: Capriccio: scalable threads for internet services. In *Proc. of the 19th ACM symposium on Operating systems principles*, pages 268–281.

Culler, D. E., Goldstein, S. C., Schauser, K. E., and von Eicken, T. (1993). Tam – a compiler controlled threaded abstract machine. In *Journal of Parallel and Distributed Computing Vol.18*, pages 347–370.

E.A.Thomas and et al (1991). Scheduler activation: Effective kernel support for the user-level management of parallelism. In *Proc. of the 13th ACM Symp. on OS Principles*, pages 95–109.

Kusakabe, S. and et al (1999). Parallel and distributed operating system cefos. In *IPSJ ISG Tech. Notes, Vol.99, No.251*, pages 25–32.

McVoy, L. and Staelin, C. (1996). lmbench: Portable tools for performance analysis, http://www.bitmover.com/lm/lm-bench.

Petterson, M. (n.d.). Perfctr, http://user.it.uu.se/ mikpe/linux/perfctr/.

Purohit, A. and et al (2003). 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.

Snavely, A. and Tullsen, D. (2000). Symbiotic jobscheduling for a simultaneous multithreading processor. In *9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 234–244.

Taniguchi, H. (2002). Drd: New connection mechanism between internal kernel and external kernel. In *Tran. of IEICE, Vol.J85-D-1, No2.*