Enhancing the performance of process level redundancy with coprocessors in symmetric multiprocessors

Tong Zhang, Hongjun Dai*, Changsong Wu and Zhiping Jia

Department of Computer Science and Technology,
Shandong University,
Box 94, No. 1500, Shunhua Road,
Jinan, Shandong, China
Fax: +86-531-8839-2498
Email: tongzhang.chn@gmail.com
Email: dahogn@sdu.edu.cn
Email: wuchangsong9001@gmail.com
Email: jzp@sdu.edu.cn
*Corresponding author

Abstract: Transient faults are rising as a crucial concern in the reliability of computer systems. As the emerging trend of integrating coprocessors into symmetric multiprocessors, it offers a better choice for software-oriented fault tolerance approaches. This paper presents coprocessor-based process level redundancy (PLR) which makes use of coprocessors and frees CPU cycle to other tasks. The experiment is conducted by comparing the performance of one CPU version of PLR and one coprocessor version PLR using a subset of optimised SPEC CPU2006 benchmark. It shows that the proposed approach enhances performance by 32.6% on average. The performance can be enhanced more if one application contains more system calls. This common technique can be adapted to other software-based fault tolerance as well.

Keywords: fault tolerance; symmetric multiprocessors; SMPs; process-level redundancy; coprocessor.

Reference to this paper should be made as follows: Zhang, T., Dai, H., Wu, C. and Jia, Z. (2018) 'Enhancing the performance of process level redundancy with coprocessors in symmetric multiprocessors', *Int. J. Computational Science and Engineering*, Vol. 16, No. 1, pp.1–8.

Biographical notes: Tong Zhang received his BS in Software Engineering at Shandong University. He is interested in micro-architecture and the operating system supports. He is the contributor/author of many open source softwares.

Hongjun Dai received his PhD in Computer Science at Zhejiang University. He is a Lecturer at the Department of Computer Science and Technology, Shandong University, China. His research interests are related to reliability of computer architecture, and cyberphysical systems. He has published more than 30 research papers at national and international journals, conference and proceedings.

Changsong Wu is a graduate student at the Department of Computer Science and Technology, Shandong University, China. He is interested in the optimisation of the operating system.

Zhiping Jia received his PhD in Control at Shandong University. He is a Full Professor at the Department of Computer Science and Technology, Shandong University, China. His research interests are related to embedded systems, and wireless sensor networks. He has published more than 60 research papers at national and international journals, conference and proceedings.

This paper is a revised and expanded version of a paper entitled 'Coprocessor as the reliability validation to enhance the performance of process level redundancy in CMPs' presented at CSS 2013, Zhangiiajie, China, 16–18 July 2013.

Copyright © 2018 Inderscience Enterprises Ltd.

1 Introduction

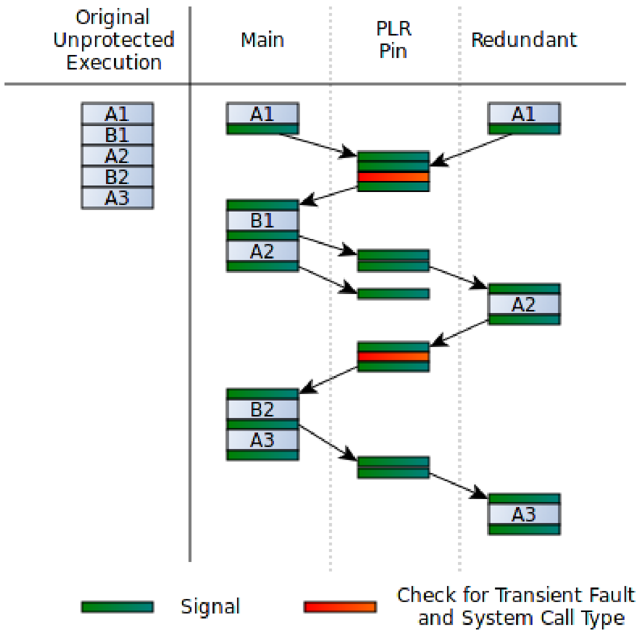
Transient faults, known as soft errors, are rising as a crucial concern in the reliability of computer systems (Michalak et al., 2005). They may occur when electronic components invert their states owing to interference by an undesirable event (Rahmouni et al., 2013). Software approaches that guarantee reliability of computer systems have many advantages since they does not need special hardware and some implementations can run on mainstream hardware without modification, but many of them use a redundant validation routine which requires additional computation resources and competes for CPU time with other processes. This paper proposes a novel usage of coprocessors (Goddeke et al., 2009) which has been widely equipped in mainstream processors and is very suitable for doing redundant validation jobs.

Dual execution at process level is one method for detecting soft errors, it is achieved by creating redundant processes and validating result of the processes at some point. These processes must take in exactly the same input for they have the same behaviour. Validation of the processes is also one main concern for that the content to be verified, the time to verify and the means by which to verify have to be chosen carefully.

There are many software approaches which aim at guaranteeing the reliability of computer systems by dual execution of processes. One of the most commonly studied methods is process level redundancy (PLR) (Shye et al., 2009). PLR creates one leading main process and multiple redundant processes of an application process and guarantees the correct execution of the application mainly by feeding processes with replicated inputs and comparing the output of them. However, the redundant validation job requires stand alone processor to execute which can cause overhead by increasing system time and it is not economically efficient. An example is shown in Figure 1. An application usually consists of two types of code blocks, one type of code blocks does not invoke system call, which are labelled A[N], another type of code blocks will invoke system calls, which is labelled B[N]. The code blocks in the application will be executed as programmed if there is no protection by PLR. Situation is much different when running PLR. Runtime speculative software-only fault tolerance (RSFT) uses ptrace to implement PLR fault-detection which reduces detection overhead by 13.36%.

In the novel computer architecture, processor vendors have put symmetric multiprocessors (SMPs) into market for a long time. SMPs brings more computation power to users, however, most of the programmes are not ready with SMPs (Donath et al., 2008). SMPs emerging these years bring a new trend in software-based approaches. Cores can be scheduled to run different jobs, for example, some cores can be organised to run redundant processes and the rest of the cores are reserved for non-redundant processes (Subramanyan et al., 2010). PLR can take advantages of SMPs by running redundant processes concurrently on multiple cores, but it still decreases the throughput of system. Meanwhile, coprocessors are commonly equipped in mainstream processors. Industries have been integrating them in their products and providing SIMD instructions which can greatly accelerate the speed of computing. For example, Intel has been delivering SSE and AVX in their processor family.

Figure 1 Execution plane of PLR, A[n]s indicate the programme code which do not invoke system call, B[n]s indicated the programme code which invoke system call (see online version for colours)


Original Unprotected Execution
Main
PLR Pin
Redundant
A1
B1
A2
B2
A3
Signal
Check for Transient Fault and System Call Type

In order to achieve higher performance and lower CPU consumption of software-based fault tolerance approaches, we propose a novel method to reduce the cost overhead by using idle cycles of coprocessor to take over validation jobs. To utilise the full power of SMP, redundant processes are dispatched and scheduled to run on different cores to speed up the execution. We later compare the result by using SPEC CPU2006 benchmark suit. Our experiment shows that the performance of fault tolerance approaches can be enhanced by offloading validation jobs to idle computing resources. Also, by offloading validation jobs to coprocessors, the freed CPU cycles can be utilised by redundant processes which in turn reduces the execution time. The experimental results show that when running selected items from a set of optimised SPEC CPU2006 benchmarks on a real SMP machine, the proposed approach reduced overhead caused by validation routine by 32.6% on average.

The rest of this paper is organised as follows: Section 2 describes related work. Section 3 describes the proposed model. Key methods used in the model are described in Section 4. Section 5 is devoted to describe and analyse experiments and results on original PLR and coprocessor version PLR. A discussion is presented in the last subsection. Conclusion is described in Section 6.

2 Related work

2.1 Software approaches to transient faults

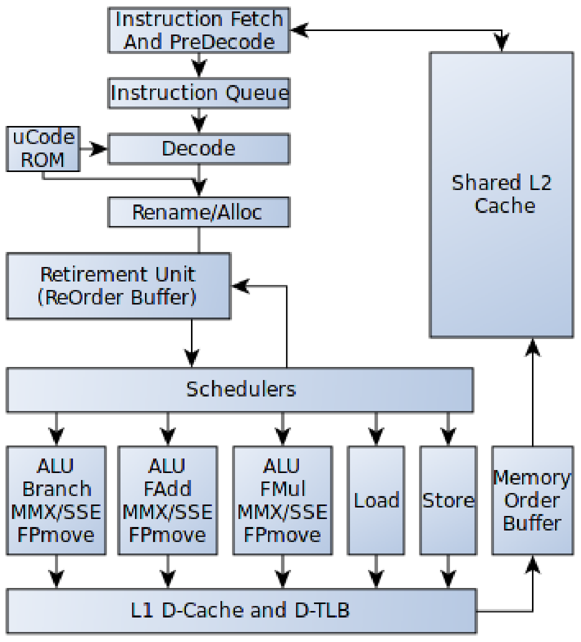
The sphere of replication (SoR) (Reinhardt and Mukherjee, 2000) is commonly used to describe the domain of redundancy and the boundary for fault detection. All executions within the SoR is repeated, all input enter the SoR is replicated and any data is compared for fault detection before leaving the SoR. All executions inside the SoR are ensured to be correct, all other executions outside the SoR are not covered. Therefore, it must be guaranteed by other means. PLR (Shye et al., 2009) is a software approach to transient fault tolerance for multi-core architecture. It creates a set of redundant processes for each original process and compares the output of the processes to guarantee correctness of execution. A syscall emulation layer is employed to take over processes' syscall request and by feeding them exactly the same data through syscall, it guarantees the same input of different redundant processes. PLR defines output as system call parameters which usually consist of data or pointers to data block. These data are collected and saved to a shared memory region and compared later. Larger data block is not copied but the CRC32 of it will be calculated and compared to avoid cost of memory copy operation and reduce memory usage. However, the comparison is done by main processor which will increase systime.

RSFT (Zhang et al., 2012) is similar to PLR, it uses ptrace to implement monitor programme instead of Intel Pin Tool which is used in PLR to avoid high runtime overhead. The monitor programme first loads the programme file and creates multiple instance of the application then traps every system call from these processes to record the parameters which are compared later for redundancy validation. CPU-aware PLR in multi-core (Aliee et al., 2011) is proposed for scheduling processes dynamically. It employs a light process to compute CPU usage periodically and determine the number of replicas. By employing a hardware-aware Microvisor (Khan and Kundu, 2010), a system can dynamically reallocate threads according to their computational resources requirement. Based on the fact that cores have different performance, threads will be dynamically scheduled to the core which will best fit the current demand. Automatically offloading computations tasks to coprocessors has been achieved on cell processors (Hoffmann et al., 2010), which automigrates programme procedure between main processors and coprocessors. The performance of applications is enhanced by the proposed method (Hoffmann et al., 2010). Multiplexed redundant execution (MRE) (Subramanyan et al., 2010) is a hardware approach which partitions into three different pools of cores.

2.2 Coprocessors as important resources

Figure 2 shows the architecture of Intel Core Processor (Doweck, 2006) which has three SSE units. The SSE units provide SIMD instructions which can complete operation on two 128 bit registers in one cycle. Coprocessors can run concurrently with main processor which provides a fast data processing path. Software optimised for them can perform better than non-coprocessor optimised version. However, they are only used by optimised software such as. In other words, it will have more idle cycles if the software uses less of it. Since coprocessor can run concurrently with main processor, the validation job can be done without stopping the processes running on the main processor. SIMD instructions provided by them are also very suitable for doing redundant validation jobs. For example, the PCMPEQQ instruction provided by Intel processors can complete comparison of two 128 bit Packed Qword Data at once which is faster than the comparison of two 64 bit data on main processor and can also be used to speed up PLR.

Figure 2 Intel core architecture (see online version for colours)



2.3 Motivations

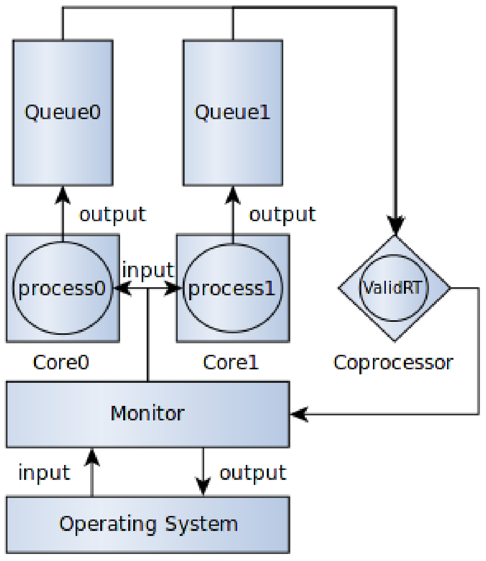
Based on our observation, by distributing redundant process among SMP cores, coprocessors can be exploited to enhance the performance of fault detection of PLR by accelerating the validation routine. We employ coprocessors to take over validation jobs including comparison routine and checksum generation routine for block data.

3 Model

3.1 Main architecture

The proposed architecture is shown in Figure 3. It can load unmodified application binary code and inspect the application process at run-time. In order to achieve this goal, a tool application is made. First, the tool is started by operating system creating a monitor process, then it loads application binary and libraries into the memory. It will create two redundant processes for the application and take control of the application before they start to execute. The redundant processes are assigned to different cores to achieve processor affinity. The monitor is employed to take care of process replication, input replication and merging output to the operating system. The replicated processes run concurrently on cores while the outputs are trapped by monitor then sent to the corresponding queue for comparison. The ValidRT process is created by monitor process and runs on the coprocessor in response for validation job. The monitor process looks after the queues. Once data is ready in queues, the ValidRT process will take care of the data and compare them byte by byte for any difference. The result is then sent to the monitor to interface with the operating system if the two queues agree with each other. Otherwise the inconsistency is reported to the monitor for further treatment. The result of input system calls are replicated and forwarded to the processes. System calls that are invoked by different processes at the same point of execution and are classified as output are merged and forwarded to the operating system. In one word, all the system calls are first verified by the ValidRT process then forwarded to the operating system. The replicated input and merged output system calls are done only once to keep the result correct.

Figure 3 Architecture overview (see online version for colours)



3.2 Dynamic scheduling of redundant processes

The monitor process controls the whole redundant system. It is designed to have the highest priority for that it does I/O task most of the time and requires little CPU time. It will wake up under circumstances when the compare queue are ready, when a system call is responded or when a transient fault is detected. It also controls the execution of the redundant processes and take care of the priority of the processes.

The ValidRT process has real-time or highest priority for it is a crucial part of the system. When a system call is invoked, it is expected to be responded as soon as possible. The valid process is expected to finish its job as quickly as possible also for that it will not block the real invoke of system calls for too much of time.

Once redundant processes are created and put into the task queue, they will be scheduled by the operating system to run on different processors as they have been pinned to designated cores. When a redundant process is blocked by a system call it will be scheduled to the waiting task queue. The ValidRT process continue running as the queues showed in Figure 3 are not empty. Once the result is returned from ValidRT, monitor process is waken up to deliver system calls to operating system and wait until system call returns. The result of system call is then forwarded to the sleeping redundant processes. These processes are immediately put into ready task queue for next scheduling from operating system. When redundant process runs out of time slice, it is scheduled off the CPU but still in the ready task queue. Other parts work almost the same as described above when a redundant process used up its time slice.

As different applications may require different service quality, a task priority value is added to the process. If one application requires better service quality, higher priority is set for it. This will increase time slice of the process as well as increasing the chance to be selected when operating system reschedule processes.

3.3 Load balancing on coprocessor

In order to utilise coprocessor in a graceful manner, the ValidRT process will sleep and wait until the queues are ready with data. Therefore, ValidRT process can be scheduled off the coprocessor to release computation resource for other applications. Taking a more extreme method, dynamically create/destroy the ValidRT process will also release the memory it occupies but result in additional time overhead.

If one system uses coprocessors heavily, such as running coprocessor consuming applications, the ValidRT will get fewer cycles to execute. This will lead to longer latency in finishing validation job and decrease of service quality. The situation can be even worse since the cache line and pre-fetch cache would also be invalided due to context switching on coprocessor. In this case, the monitor process should keep an eye on coprocessor usage. A start threshold and a stop threshold are set. Once the usage of coprocessor is below start threshold, the ValidRT process will be scheduled on the coprocessor. If the usage of coprocessors is above stop threshold, the ValidRT process will be scheduled on the main processor. Considering a CPU with two or more coprocessors or many different types of coprocessors which can be used to run ValidRT process, the process can be migrated from one busy coprocessor to a free coprocessor if the situation described above happens. However, performance can still degrade if frequent migration happens, plus the heavy load on coprocessors and main processors.

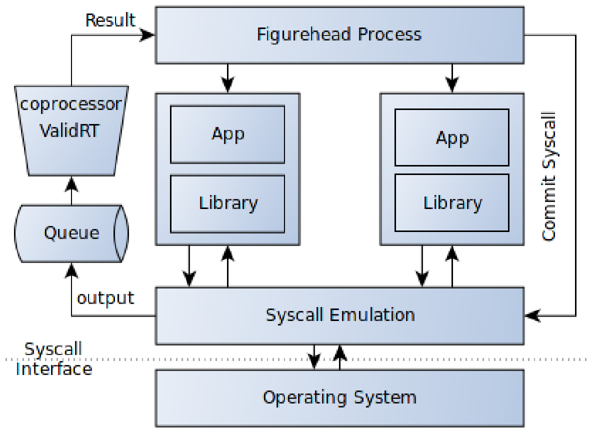
4 Key methods

4.1 Detailed model

In order to measure the performance of the proposed approach, we use a slightly different version of PLR as shown in Figure 4 which eliminated other not relevant modules such as fault recovery module and watchdog module. At the very beginning, the figurehead (monitor) process is launched. It initialises resources such as FIFO queue and pipe which is used to store output data and communicate with system call emulation layer, coprocessor's ValidRT routine and system call emulation layer. It then loads the application into the memory, creates two redundant processes which run concurrently on different cores and takes control over them before they start to execution. The system call emulation layer traps every system call whenever the redundant processes call and collects the information for comparison. When queues are checked by figurehead process, the output from processes is pushed to the corresponding queue. The ValidRT process is woken up or created for validation whenever the queues are ready. System call request is then forwarded to the operating system for execution if there is no disagreement. Once the system calls return, the results are immediately forwarded to the redundant process.

Compared to PLR, our approach utilises coprocessors for the following two aspects: First, fast checksum calculation for data block. If figurehead process encounters pointers to data block, it will call ValidRT process to calculate the Checksum of the data block using dedicated instructions which is faster than that of main processor. System benefits from it for it saves memory copy time and compare time. Second, fast comparison for output queue. Data from the queue can stream into the coprocessor and results come out non-stopping. If the coprocessor is equipped with SIMD instructions, this will be even faster.

Figure 4 PLR system architecture with two redundant processes (see online version for colours)



4.2 Coprocessor version

In Algorithms 1 and 2, the letter *N* denotes system call number, letter *P* denotes parameter of the system call, *PID* stands for PID of the process. Flag *error* is set and reported when an inconsistency is detected. The *queue* refers to the queue illustrated in Figure 3 which is distinguished by PID.

Algorithm 1 Validation routine using coprocessors

```
1:  procedure Validation-cop($N$, $P$, $PID$)
2:      $error \leftarrow 0$
3:      $queue(PID) \leftarrow N$        ▷ Push syscall number to queue
4:      for all $p \in V(P)$ do
5:          if $p$ is Pointer then
6:              CHECKSUM_THREAD($p$, $queue$, $PID$)        ▷ Offload
7:          else
8:              $queue(PID) \leftarrow p$        ▷ Push parm to queue
9:          end if
10:     end for
11:     Join_CHECKSUMTHREAD( )        ▷ Wait for Checksum Calculation
12:     if queue is ready then
13:         DIFF_THREAD($queue$, $error$)        ▷ Offload
14:     end if
15:     Join_DIFFTHREAD( )        ▷ Wait for Comparison
16:     if $error \neq 0$ then
17:         raiseup(error)        ▷ Error Occured
18:     end if
19: end procedure
```

Algorithm 2 Validation routine using main processors

```
1:  procedure Validation-noncop($N$, $P$, $PID$)
2:      $error \leftarrow 0$
3:      $queue(PID) \leftarrow N$        ▷ Push syscall number to queue
4:      for all $p \in V(P)$ do
5:          if $p$ is Pointer then
6:              CHECKSUM($p$, $queue$, $PID$)        ▷ Calculate Checksum on CPU
7:          else
8:              $queue(PID) \leftarrow p$        ▷ Push parm to queue
9:          end if
10:     end for
11:     if queue is ready then
12:         $error \leftarrow$ DIFF($queue$)        ▷ Compare
13:     end if
14:     if $error \neq 0$ then
15:         raiseup(error)        ▷ Error Occured
16:     end if
17: end procedure
```

The proposed validation routine is designed as shown in Algorithm 1, naming main thread. When a system call is caught, the validation routine is called on behalf of collecting system call type and parameters as well as comparing for disagreement. The routine first pushes the system call number to the queue, then traverses parameters to find out pointers that point to large data block. It is pushed into the queue directly if one parameter is not a pointer. Otherwise, the CHECKSUM_THREAD is called to calculate the checksum of the block that will accomplish fast checksum calculation. After all information is collected, a DIFF_THREAD is created for comparison that will accomplish fast comparison for output queue. All CHECKSUM_THREAD and DIFF_THREAD run concurrently on coprocessors with the main thread which runs on main processor.

4.3 Main processor version

The original validation routine is designed as shown in Algorithm 2. The difference between the two versions is that the original version uses main processor to calculate checksum and compare for disagreement. The routine is designed to execute serially on the main processor. The number of system calls and pointer type parameters must be taken into consideration that these factors affect greatly the run time of validation routine. According to the algorithm, the more system calls requested the more time is needed for validation. The same circumstance is also true to pointer parameters since more operations are required for calculating checksum of data block.

This approach as well as PLR can only trap I/Os which go through system calls, because that there are lots of other means of I/O such as mmap which maps file to memory region and I/O is completed by reading/writing the mapped memory regions. Another example is DMA where data are transferred between memory and device directly. It also uses mmap to map the devices' address to user-space applications' memory region to enhance the performance of application. It means that only a subset of I/O is monitored and checked. Soft errors occur within these methods are left undetected. Read/write operations to these memory regions must be checked just like the way it do to system calls to cover these errors.

5 Experiments and results

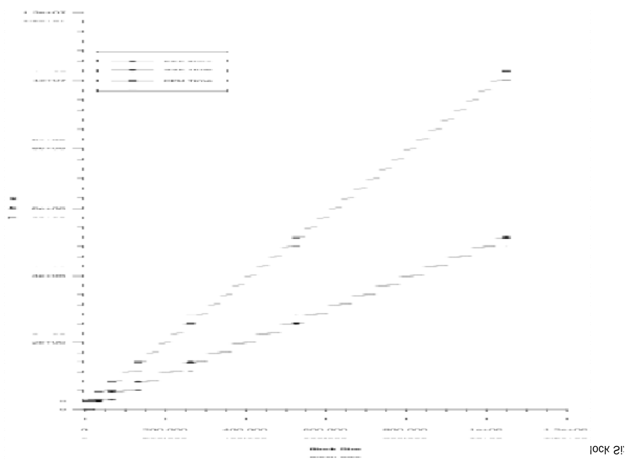
5.1 Experiment setup

The experiment is conducted using the setup with an Intel dual core processor P7450 with SSE2 extension, 3,072 Kbyte L2 cache, 4 GB system memory, and Gentoo Linux (kernel version 3.9.4, gcc 4.6.3, 64 bit kernel and multi-lib userland). Intel Pin Tool is used to make a custom tool programme to inspect instructions and inject validation code. The performance is measured by run time of application. Selected benchmarks of SPEC CPU2006 are tuned with-O2-march=native option to produce 64 bit target and the run time of validation routine is measured using time tool. Two versions of compare routines are implemented including a PLR-based validation routine without coprocessor support called NON-COPRO and a coprocessor version called COPRO which uses SSE2 SIMD extension. CRC32 is employed as checksum method in both versions. All measurements were repeated ten times and the average values are taken as input for calculation.

5.2 Check routine

The performance of validation routines is first calculated to ensure the effectiveness. The two routines are fed with the same set of data which contains different size of data block varies from 512 B to 1 GByte, in each test case the block size is twice the previous size. The time consumed by the validation routine is measured using gettimeofday for accurate time counting. The result is compared in Figure 5. The coprocessor version of validation routine is about two times faster than CPU version in all block tests. This proves the effectiveness of the coprocessor version validation routine.

Figure 5 Time consumption of the routines

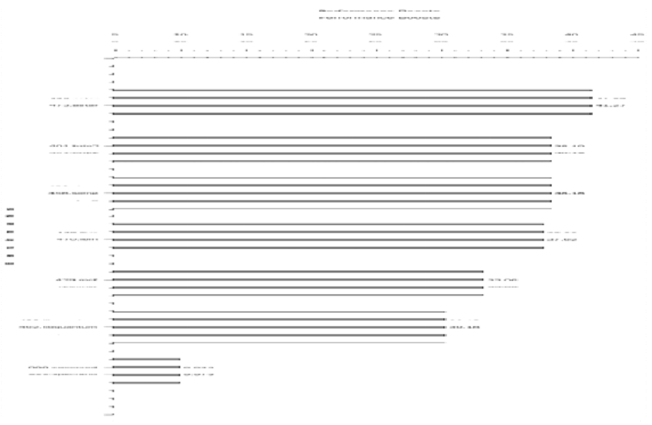


5.3 Performance evaluation

$$Per\ f\ Boost = \frac{t_{NON\_COPRO} - t_{COPRO}}{t_{NON\_COPRO}} \cdot 100 \qquad (1)$$

The performance boost is calculated by using measured run time of two versions of PLR and formula 1. This index is used in the reset of the paper to find out relations of different factors which may affect performance. As is shown in Figure 6, all of applications' performance get enhanced by using coprocessors. The most significant is 473.astar that is enhanced by 41.27%. The least significant is 999.specrand that is only enhanced by 9.973%. The performance is enhanced by 32.6% on average which is very impressive.

Figure 6 Performance boost

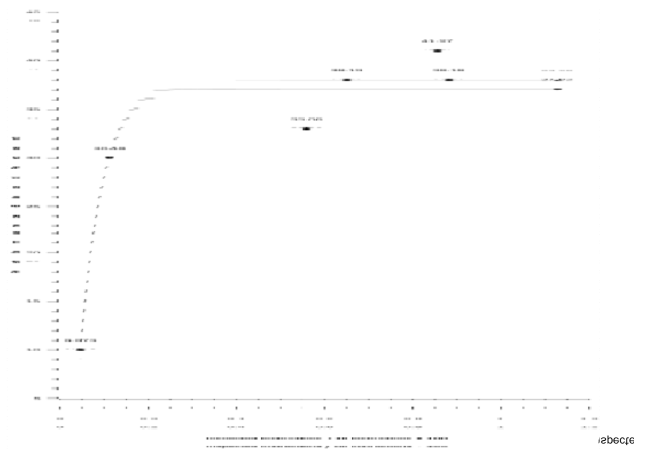


5.4 *Impact of system call ratio*

$$I / A = \frac{N_{SyscallInstructions}}{N_{AllInstruction}} \cdot 100 \qquad (2)$$

Supposing performance should boost as the proportion of system calls increase, the instructions which cause system calls are counted to compare with the performance boost. Figure 7 shows the relation between percent of instructions inspected (I/A) for the selected benchmark and performance boost. Especially, 999.specrand has 0.04 I/A and gets least performance boost. The curve rises up until about 0.65 I/A and tends to be slow rising after then. This indicates that the performance get enhanced very quickly with the increase of I/A. After some point, the curve tends to be slower increasing which indicates that there must be other factors that affect overall performance boost.

Figure 7 Percent of instructions inspected and performance boost



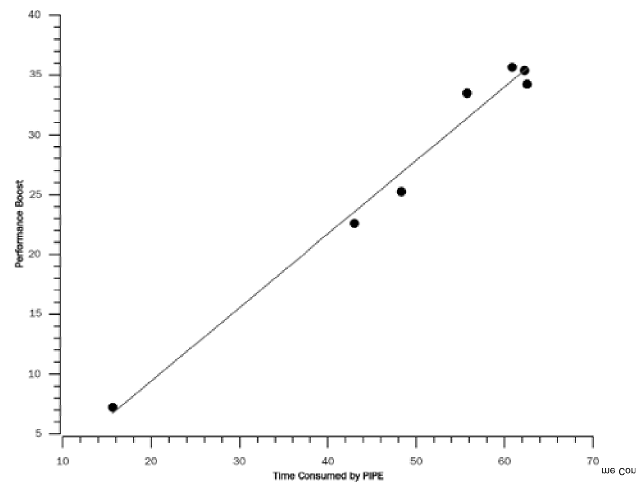
As the arbitrary behaviour of any unknown applications, instructions which are not inspected may take arbitrary time to execute and even make use of coprocessors. This can actually affect the overall performance boost since it is calculated by formula 1. Figure 7 takes I/A into consideration which can be counted before the application runs. In this case, the performance boost of some applications can be predicted before the application actually runs.

5.5 *Relation with pipe time*

The validation routine uses pipe to implement FIFO queue for communication with figurehead process. Time consumed by pipe also has a direct relationship with the number of system calls. So, there may be a chance that pipe time can be used to predict performance boost. To see the relation between the number of system calls and performance boost more clearly, the time used to pass data through pipe is measured to compare with the performance boost. It can also help to find out whether pipe blocks the increasing of performance.

As shown in Figure 8, the performance boost has strong linear relationship with time consumed by pipe since the time is linear to data pass through it. So, the performance boost can be predicted by the amount of data that flows through the pipe. Furthermore, Figure 8 shows no sign of bottle neck related to pipe. But using other low latency IPC methods such as shared memory (SHM) can help improve the performance. When testing with typical 512 byte chunk of data this pipe latency is about 7˜50 us, 15 us on average, while the latency of SHM is about 6 us on average.

Figure 8 Pipe time and performance boost



Since the benchmarks selected from SPEC CPU2006 do not utilise coprocessors even when optimised with additional gcc option, Figure 8 shows an approximate linear relationship between performance boost and pipe time, which in fact indicates the performance boost of validation routine. This is true only when coprocessors are not occupied by other applications.

5.6 *Discussion*

The experiment shows that the time used for comparison is reduced by 32.6% on average. Applications with more system calls and larger data input/output benefit more from this approach compared to original PLR, because there are more data to be processed by compare routine. The performance boost can be predicted using Figure 8.

One drawback of this approach is that if one application is designed to use coprocessors or optimised by compilers to use coprocessors, the fault detection routine will compete with applications for coprocessor which will lead to degrade in performance. If this circumstance happens, another mechanism is required to detect the race condition and report to the figurehead application. The figure head application can then migrate the routine to other spare computation resources such as main processors to avoid such competition.

Limited fault coverage is another drawback, as not all of the I/O operations go through system calls. Other means of I/O such as mmap operate very differently from the studied method. One possible solution to this problem is to monitor read/write operations to the memory region used by these I/O methods and translate these operations to system calls, then it can be easily handled by current approach. Since mmap is usually used to operate on frequently used and large data chunk, it can predicted that this will make heavy use of compare routine and the routine should be fast enough in order not to slow down the system. The method mentioned above may not be efficient enough to handle such large amounts of data since the process will stop running when it waits for the ValidRT process. Situation can be even worse with frequent access to the mmaped region.

6 Conclusions

This paper presents a novel usage of coprocessors that exploit under utilised computation resources to take over validation jobs in PLR. The use of coprocessors frees CPU cycles to other tasks. The experimental results show that when running selected items from a set of optimised SPEC CPU2006 benchmarks on a single SMP machine, the proposed approach reduced overhead caused by validation routine by 32.6% on average. The same technique can also be applied to other software-oriented fault tolerance approaches.

References

Aliee, H., Zarandi, H.R. and Tajary, A. (2011) 'CPU-aware, process-level redundancy to tolerate faults in multicore', in *2011 International Conference on High Performance Computing and Simulation (HPCS)*, IEEE, pp.343–349.

Donath, S., Iglberger, K., Wellein, G., Zeiser, T., Nitsure, A. and Rude, U. (2008) 'Performance comparison of different parallel lattice Boltzmann implementations on multi-core multisocket systems', *International Journal of Computational Science and Engineering*, Vol. 4, No. 1, pp.3–11.

Doweck, J. (2006) *Inside Intel Core Microarchitecture and Smart Memory Access*, Intel White Paper.

Goddeke, D., Wobker, H., Strzodka, R., Mohd-Yusof, J., McCormick, P. and Turek, S. (2009) 'Co-processor acceleration of an unmodified parallel solid mechanics code with feastgpu', *International Journal of Computational Science and Engineering*, Vol. 4, No. 4, pp.254–269.

Hoffmann, R., Prell, A. and Rauber, T. (2010) 'Dynamic task scheduling and load balancing on cell processors', in *2010 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, IEEE, pp.205–212.

Khan, O. and Kundu, S. (2010) 'Thread relocation: a runtime architecture for tolerating hard errors in chip multiprocessors', *IEEE Transactions on Computers*, Vol. 59, No. 5, pp.651–665.

Michalak, S.E., Harris, K.W., Hengartner, N.W., Takala, B.E. and Wender, S.A. (2005) Predicting the number of fatal soft errors in Los Alamos National Laboratory's ASC Q supercomputer', *IEEE Transactions on Device and Materials Reliability*, Vol. 5, No. 3, pp.329–335.

Rahmouni, K., Chabanet, S., Lambelin, N. and Petrot, F. (2013) 'Design of a medium voltage protection device using system simulation approaches: a case study', *International Journal of Embedded Systems*, Vol. 5, No. 1, pp.53–66.

Reinhardt, S.K. and Mukherjee, S.S. (2000) 'Transient fault detection via simultaneous multithreading', in *ACM SIGARCH Computer Architecture News*, ACM, Vol. 28, pp.25–36.

Shye, A., Blomstedt, J., Moseley, T., Reddi, V.J. and Connors, D.A. (2009) 'Plr: a software approach to transient fault tolerance for multicore architectures', *IEEE Transactions on Dependable and Secure Computing*, Vol. 6, No. 2, pp.135–148.

Subramanyan, P., Singh, V., Saluja, K.K. and Larsson, E. (2010) 'Multiplexed redundant execution: a technique for efficient fault tolerance in chip multiprocessors', in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, pp.1572–1577.

Zhang, Y., Ghosh, S., Huang, J., Lee, J.W., Mahlke, S.A. and August, D.I. (2012) 'Runtime asynchronous fault tolerance via speculation', in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ACM, pp.145–154.