JCSE Journal of Computing Science and Engineering

Regular Paper

Journal of Computing Science and Engineering,
Vol. 5, No. 4, December 2011, pp. 283-287

A Technique for Fast Process Creation Based on Creation Location

Byungjin Kim and Youngho Ahn
Department of Electronics Computer Engineering, Hanyang University, Seoul, Korea
idangoon@gmail.com, ghduddks84@gmail.com

Ki-Seok Chung*
Department of Electronic Engineering, Hanyang University, Seoul, Korea **kchung@hanyang.ac.kr**

Abstract
Due to the proliferation of software parallelization on multi-core CPUs, the number of concurrently executing processes is rapidly increasing. Unlike processes running in a server environment, those executing in a multi-core desktop or a multi-core mobile platform have various correlations. Therefore, it is crucial to consider correlations among concurrently running processes. In this paper, we exploit the property that for a given created location in the binary image of the parent process, the average running time of child processes residing in the run-queue differs. We claim that this property can be exploited to improve the overall system performance by running processes that have a relatively short running time before those with a longer running time. Experimental results verified that the running time was actually improved by 11%.

Category: Embedded computing

Keywords: Process management; Software parallelization; Multi-core CPU

I. INTRODUCTION

Improving performance through increasing the clock speed of a single processor core is limited by high energy consumption and temperature. Thus, performance improvement utilizing multi-core processors has been actively studied [1]. Consequently, software structures should be changed to fully utilize the multi-core processor. In other words, there have been numerous researches on creating multiple threads in a single application and executing them in parallel to improve performance [2]. Therefore, the number of concurrently executing processes is steadily increasing, and it is becoming crucial to manage those processes effectively [3].

Process management under server environments has been previously studied. However, unlike processes running in a server environment, those executing in a multi-core desktop or a multi-core mobile platform have various correlations. Therefore, it is crucial to consider correlations among concurrently running processes [4].

We observe that a significant number of processes are not created directly by the user but depend on the coding style followed by the programmer when writing an application program. That is, there is an increasing trend that the process creation pattern and the process handling characteristics are not explicitly controlled by the user, but depend on the coding style used to write the application program. Especially, processes that are created and managed according to the specific style that was used to write the application are typically created at fixed locations in the binary image of the parent process. We will show that the location where the process is created in the binary image of the parent process and the average running time of child processes residing in the run-queue are strongly corre-

Open Access http://dx.doi.org/10.5626/JCSE.2011.5.4.283 http://jcse.kiise.org

Received 06 July 2011, **Accepted** 26 August 2011
*Corresponding Author

 pISSN: 1976-4677 eISSN: 2093-8020

lated. We will also present that correlations between the process creation location and the average process execution time can be utilized for efficient system management, especially for process scheduling.

The rest of the paper is organized as follows. In Section II, related works on the characteristics and requirements of multi-core desktop and multi-core mobile device environments along with the application properties running on them are presented. In Section III, process correlations are addressed and our experimental method is presented in detail. In Section IV, we analyze our experimental results. Conclusions will follow in the final section.

II. RELATED WORKS

There have been many researches on application properties under a multi-core desktop environment. Research in [3] studied the degree of thread-level parallelism when applications were executed under a multi-core desktop environment. The results were analyzed. In [5], the authors presented that the response time characteristics when applications request I/O operations under a desktop environment differ from those under a server environment. In [6], recent theoretical results for dynamic power management of multi-core processors were summarized. In [7] and [8], methods to estimate the worst-case execution time of applications running on multi-core processors were proposed.

These previous researches have either focused on analyzing runtime characteristics that can be observed after each process is forked, or considering only the parent-child relationship between processes to predict and exploit processes' runtime characteristics.

We suggest that the locations where processes are forked in the parent processes' binary image provide a new clue to help analyze and predict the child processes' runtime characteristics. And we examine how the new clue can contribute to predicting and exploiting the child processes' runtime characteristics through experiments.

III. LOCATION-BASED PROCESS MANAGEMENT

We observe that a significant number of processes are created by the coding patterns used to write an application program. Even though the number of concurrently executing processes and the execution patterns are partially dependent on the system user, they are also significantly dependent on the style in which the programmer writes the application. Consequently, it is important to understand how processes are created by applications, and what characteristics are possessed by each process.

The child processes are typically created at specific locations of the parent processes' binary image. That is, when programmers write applications, they call specific library routines or system calls to create child processes. Therefore, locations where many child processes are created can be clearly identified. We will use the name of the binary image of the parent process and the program counter value for the location where the child process is created from the image. These are attached to each child process as process attributes. We claim that the behavioral patterns are correlated with these two attributes, and we verify this claim experimentally. To be precise, we measure the average execution time of child processes residing in the run-queue, and analyze the correlation between this time and the two attributes. Through this analysis, we devise a novel process management method to improve the system performance.

Our experimental environment is based on Linux Kernel 2.6.31.6 and GNU Library C 2.11.2 and we modified these in order to understand the behavior. An off-line analysis result of the data collected from this environment is stored in a table so that the kernel operation can refer to it. By utilizing the information collected by the off-line analysis, the overall performance is significantly improved.

A. Process Creation of Linux Systems

Processes are created using fork(), vfork(), and clone() system calls in Linux systems. However, it is uncommon for application programs to invoke these system calls directly. Instead, these system calls are usually called indirectly via the fork(), vfork(), and clone() library functions provided by GNU Library C. In the Linux kernel, the do_fork() function is called to actually carry out the process creation task for the fork(), vfork() and clone() system calls.

B. Data Collection for Process Creation Location and The Average Execution Time

To understand the correlation between the process creation location and the process behavior, first we need to collect the information on the locations (i.e., program counter values) where the child processes are created. To this end, we modified the routines for process creation in the GNU Library C and the

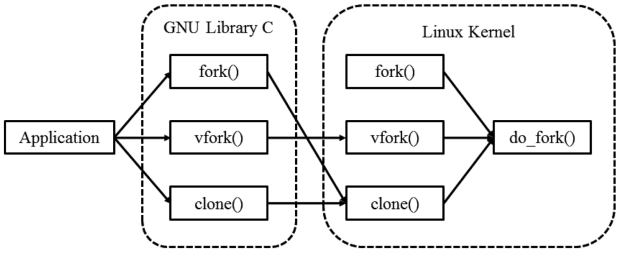

Fig. 1. Process creation in Linux system.

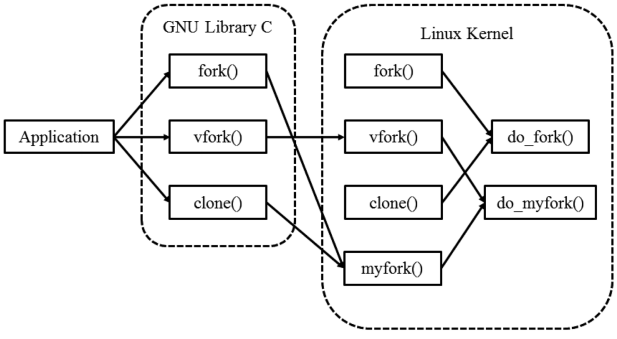

Fig. 2. Modified process for process creation.

Linux kernel. First, we collect the return addresses that are stored in the applications' stack for the fork() and clone() functions. Since vfork() passes the return address to the kernel, no modification is necessary. These return addresses correspond to the program counter values where the child process is created. The collected return addresses are passed to the Linux kernel as arguments for system calls, and stored in a data structure.

To pass return addresses, system call interfaces should be changed. However, changing the interface of the original system calls can be a very dangerous operation, which may cause the system operation to become unstable. Therefore, we implemented a new wrapper function called myfork(), so that this wrapper function is called instead of the original one when we collect the information. We also add a new wrapper function for do_fork() called do_myfork(), so that the myfork() and vfork() system calls will call the wrapper for do_fork(). Function do_myfork() stores the collected data, which consist of the name of the binary image of the parent process and the program counter value indicating the location of the child process creation in the image, in the data structure. It also performs the original function of creating processes. A simple routine is added to the system calls fork() and clone() and function do_fork() that enables them to print out kernel error messages every time that the calls are invoked under exceptional circumstances. In this manner, the original do_fork() is called when the calling sequence fails to follow the proposed modified process creation path. In this case, we can identify the exceptional cases by the printed kernel error message.

Also, we need to collect the average runtime, which is defined as the time during which each process is residing in the run-queue. We collect the information by modifying the process scheduler of the Linux kernel, which has a scheduler called completely fair scheduler (CFS). In CFS, each run-queue is implemented using a red-black (RB) tree structure [9]. The average execution time of each process can be obtained by computing the average duration time between the time when the process is inserted into the RB tree and the time when the process is removed from the RB tree. The Linux kernel calls the update_curr() function to compute the actual execution time of the current process on the processor. In our experiment, we modified the update_curr() function to store the execution times in a separate data structure in addition to performing the original function.

Using all the collected data of the binary image, the program counter values, and the average execution times, we can compute the average time that each child process resides in the run-queue after it was created at a specific location of the binary image of the parent process.

C. Utilization of Correlation between the Process Creation Location and the Average Execution Time

We claim that depending on the location of the child process creation, the average execution time can differ significantly, and we used the collected data to verify this property. In the Linux scheduler, a process with a relatively short execution time is assumed to have a more interactive property than that with a large execution time. Hence the scheduler assigns a higher priority to such processes in order to improve the overall response time. In reality, the CFS incorporates the actual execution time of each process into the process priority and computes the virtual runtime after normalization. After computing the virtual runtime, on scheduling a process, it selects the process with the shortest virtual runtime for execution. However, this algorithm has a limit that even though an interactive process is created, it can only get CPU-time if it has the smallest computed virtual runtime.

On the other hand, the proposed method in this paper is based on an offline analysis on the average execution time. If the location where processes are created in the binary image of the parent process and the average running time of the child processes residing in the run-queue are strongly correlated, we could exploit the correlation to predict the interactivity of the newly created child processes at each location. Thus, when interactive processes are likely to have a shorter runtime, based on the analysis they may be scheduled to the CPU before CFS, thus shortening the overall response time. From our experiments, we verified that such interactive processes are executed immediately after creation, without waiting until they have the smallest virtual runtime. Thus, the overall performance is improved.

IV. EXPERIMENTAL RESULTS

A. Data Analysis

Our experimental environment is a virtual machine environment running on a platform that consists of Intel Core i7-2600 3.40GHz, 4GB of RAM, Windows 7 (32bits). Data collection and scheduler modification are done on Linux Kernel 2.6.31.6 and GNU Library C 2.11.2. We used a set of applications on Firefox [10] consisting of streaming audio play, web game play, and webpage surfing. In addition to the experiment on Firefox, we also carried out experiments on the Make utility and OpenOffice [11].

To analyze the correlation between the location of the process creation and the average runtime, first we ran the Firefox web browser on a Linux system after we modified the system as explained.

Table 1 summarizes the results. The set of locations for child process creation differs according to the running application, and the average execution times differ according to the values of the program counter. Also, we can verify from the case of the binary image called *run-mozilla.sh* that even though a child process is created from the same location, the average execution time differs significantly depending on the type of task that the web browser executes.

Next, we compiled the Linux kernel with the Make utility. Table 2 shows the results. As in the cases of the web browser, the average execution times differ according to the program counter and the binary image name.

Finally, Table 3 shows the results obtained from running OpenOffice. We ran a spreadsheet program (Calc), a 2D painting program (Draw), and a word processing program (Writer) in OpenOffice. In the spreadsheet program, we entered some numbers and then calculated the average and the standard deviation of the data. In the 2D painting program, we drew simple figures and colored them. In the word processor program, we typed a

Table 1. The average execution time for process creation location on Firefox

Test	Binary Image Name	Program Counter	Avg. Runtime (ns)
Streaming Music Play	run-mozilla.sh	08080b74	310871
	run-mozilla.sh	b77ccc9d	126086
	firefox	b77d6f34	160484
	firefox	b7820c9d	231821
Web Game Play	run-mozilla.sh	08080b74	2052582
	run-mozilla.sh	b7888c9d	39135
	firefox	b7892f34	1753332
	firefox	b76fdc9d	134240
Webpage Surfing	run-mozilla.sh	08080b74	8546944
	run-mozilla.sh	b770dc9d	31877

Table 2. The average execution time for process creation location on theMake utility

Test	Binary Image Name	Program Counter	Avg. Runtime (ns)
Make	make	08050ef4	199742
	make	080549e9	320698
	sh	08080b74	842219
	gcc	08062bc6	7253607
	gcc	0804c3ab	509999

Table 3. The average execution time for process creation location on OpenOffice

Test	Binary Image Name	Program Counter	Avg. Runtime (ns)
Calc	soffice	08080b74	1083365
	soffice	b771dc9d	77464
	scalc.bin	b779797e	1334359
Draw	soffice	08080b74	659877
	soffice	b76e1c9d	69569
	sdraw.bin	b775b97e	1273092
Writer	soffice	08080b74	888827
	soffice	b7560c9d	62428
	swriter.bin	b75da97e	1309738

short text and carried out spellchecking.

We can infer from the above results that the average execution times of the child processes differ according to the creation location in the parent process's binary image.

B. Performance Evaluation

This section will address how we can utilize the correlation information to improve the overall system performance. For this purpose, we modified the scheduler such that the process with the shortest average runtime from the offline analysis of the binary image and the process creation location will get CPU-time after being assigned the highest priority immediately after its creation.

Table 4. Comparison of execution times between the conventional and proposed methods

	Before	After
1	5.80	5.39
2	6.15	5.25
3	5.90	5.60
4	6.28	5.17
5	6.06	5.32
Average	6.04	5.35
Standard Deviation	0.17	0.15

In this experiment, we mainly used web browser executions, since the web browser runs a variety of workloads under a single application [12]. Experimental results show that when streaming audio decoding and web surfing are executed concurrently, streaming audio decoding is finished earlier, since its average execution time is relatively shorter.

V. CONCLUSION

As the number of concurrently executing processes on multi-core desktops or mobile devices is rapidly increasing, it is crucial to manage such concurrent processes efficiently. Unlike processes running in a server environment, those executing in a multi-core desktop or a multi-core mobile platform have various correlations. Therefore, it is crucial to consider correlations among concurrently running processes. In this paper, we exploit the property that depending on the created location of child processes in the binary image of the parent process, the average running time of the child process residing in the run-queue differs. To verify this property we modified the Linux kernel and GNU Library C, and carried out experiments. Through the experiments, we could verify that the average execution time of the child process may differ according to the binary image of the parent process and the location of the child process creation. Also we modified the kernel scheduler such that we assign a higher priority to processes with shorter average runtime when they are invoked at specific locations. The experimental results show that when we execute streaming music play and web browsing in parallel, the average execution time of the streaming music play is shorter. This verifies that our proposed scheduling method is effective and is sufficiently accurate to achieve overall performance improvement. Lastly, we have found additional possible correlations in the process creation locations, and methods to exploit these additional correlations for optimizing process management. Especially, we are seeking to exploit the child processes' average running time according to the created location as a workload weight of each child process in order to calculate the effective workload of each core in multi-core systems.

ACKNOWLEDGEMENTS

This research was supported by the MKE(The Ministry of Knowledge Economy), Korea, under the ITRC(Information Technology Research Center) support program supervised by the NIPA(National IT Industry Promotion Agency) (NIPA-2011-C1090-1100-0010)

REFERENCES

1. S. H. Fuller and L. I. Millett, "Computing performance: game over or next level?," *Computer*, vol. 44, no. 1, pp. 31-38, 2011.
2. B. Catanzaro, A. Fox, K. Keutzer, D. Patterson, S. Bor-Yiing, M. Snir, K. Olukotun, P. Hanrahan, and H. Chafi, "Ubiquitous parallel computing from Berkeley, Illinois, and Stanford," *IEEE Micro*, vol. 30, no. 2, pp. 41-55, 2010.
3. G. Blake, R. G. Dreslinski, T. Mudge, and K. Flautner, "Evolution of thread-level parallelism in desktop applications," *Proceedings of the 37th Annual International Symposium on Computer Architecture*, Saint-Malo, France, 2010, pp. 302-313.
4. H. Sutter and J. Larus, "Software and the concurrency revolution," *Queue*, vol. 3, no. 7, pp. 54-62, 2005.
5. M. Zhou and A. J. Smith, "Analysis of personal computer workloads," *Proceedings of the 7th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, College Park, MD, 1999, pp. 208-217.
6. T. W. Kim, "Task-level dynamic voltage scaling for embedded system design: recent theoretical results," *Journal of Computer Science and Engineering*, vol. 4, no. 3, pp. 189-206, 2010.
7. J. Yan and W. Zhang, "Bounding worst-case performance for multi-core processors with shared L2 instruction caches," *Journal of Computer Science and Engineering*, vol. 5, no. 1, pp. 1-18, 2011.
8. Y. Liu and W. Zhang, "Bounding worst-case data cache performance by using stack distance," *Journal of Computer Science and Engineering*, vol. 3, no. 4, pp. 195-215, 2009.
9. R. Love, "The Linux scheduling implementation," *Linux Kernel Development*, 3rd ed., R. Love, Ed., Upper Saddle River, NJ: Addison-Wesley, 2010, pp. 50-61.
10. "Firefox," http://www.mozilla.org/.
11. "OpenOffice.org," http://www.openoffice.org/.
12. A. Taivalsaari, T. Mikkonen, D. Ingalls, and K. Palacz, "Web browser as an application platform: the lively Kernel experience," *Proceedings of the 34th Euromicro Conference on Software Engineering and Advanced Applications*, Parma, Italy, 2008, pp. 293-302.

Byungjin Kim

Byungjin Kim received his BS in Computer Science and Engineering from Hanyang University, Seoul, Korea in 2009. Since 2010, he has been pursuing an MS course at Hanyang University, Seoul, Korea. His research interests include software parallelization, operating systems, and embedded multi-core architecture.

Youngho Ahn

Youngho Ahn received his MS in Electronics and Computers Engineering from Hanyang University, Seoul, Korea in 2010. Since 2010, he has been pursuing a PhD course at Hanyang University, Seoul, Korea. His research interests include low power embedded system design, parallelization, virtualization, and embedded multi-core architecture.

Ki-Seok Chung

Ki-Seok Chung received his BE in Computer Engineering from Seoul National University, Seoul, Korea in 1989, and his PhD in Computer Science from University of Illinois at Urbana-Champaign in 1998. He was a Senior R&D Engineer at Synopsys, Inc. in Mountain View, CA from 1998 to 2000, and was a Staff Engineer at Intel Corp. in Santa Clara, CA from 2000 to 2001. He also worked as an Assistant Professor at Hongik University, Seoul, Korea from 2001 to 2004. Since 2004, he has been an Associate Professor at Hanyang University, Seoul, Korea. His research interests include low power embedded system design, multi-core architecture, image processing, reconfigurable processors and DSP design, SoC-platform based verification and system software for MPSoC.