Prediction of Optimal Readahead Parameter in Linux by Using Monitoring Tool

Ekaterina Gorelkina
SRC Moscow, Samsung Electronics
e.gorelkina@samsung.com

Sergey Grekhov
SRC Moscow, Samsung Electronics
grekhov.s@samsung.com

Jaehoon Jeong
SAIT, Samsung Electronics
hoony_jeong@samsung.com

Mikhail Levin
SRC Moscow, Samsung Electronics
m.levin@samsung.com

Abstract

Frequently, application developers face to the hidden performance problems that are provided by operating system internal behavior. For overriding such problems, Linux operation system has a lot of parameters that can be tuned by user-defined values for accelerating the system performance. However, in common case evaluating of the best system parameters requires a very time consuming investigation in Linux operating system area that usually is impossible because of strong time limitations for development process.

This paper describes a method that allows any application developer to find the optimal value for the Linux OS parameter: the optimal maximal read-ahead window size. This parameter can be tuned by optimal value in easy way that allows improving application performance in short time without getting any knowledge about Linux kernel internals and spending a lot of time for experimental search for the best system parameters.

Our method provides the prediction of optimal maximal read-ahead window size for Linux OS by using the monitoring tool for Linux kernel. Scenario of our method using is very simple that allows obtaining the optimal value for maximal read-ahead window size for the single application run. Our experiments for Linux 2.6 show that our method detects an optimal read-ahead window size for various real embedded applications with adequate accuracy and optimization effect can be about a few and even a few dozen percents in comparison to default case. The maximal observed optimization effect for accelerating the embedded application start-up time was 59% in comparison to default case.

Taking into account these facts the method proposed in this paper has a very good facilities to be widely and simply used for embedded applications optimization to increase their quality and effectiveness.

1 Introduction

In modern computing systems high performance disk drive systems very often have a serious problem related with its performance, effectiveness and speed. Usually the disk input-output (I/O) performance is related with time which needs to mechanical parts of the disk to move to a location of the data storing. This time usually defines the time delays in the data extraction. Operating systems incorporate disk read-ahead cache to minimize the time delays. Typically, the data from the disk are buffered in a memory with a relatively fast access time. If requested data is already reside in the cache memory, the data can be transferred directly from the cache memory to the requester. In result the performance is increased because the access to data from the cache memory is substantially faster than the access from the disk drive [1].

Very often such cache can be sufficiently effective. But sometimes it can produce the low system performance. This relates with the sensitivity of the read-ahead cache to cache hit statistics [2]. A read-ahead cache having a low hit rate may perform more poorly than an uncached disk due to caching overhead and queuing delays, among others. This problem is especially important to be solved for embedded systems, because they usually have less memory and slow CPUs than servers and workstations.

Performance of read-ahead subsystem can be controlled by its main parameter - maximal read-ahead window

size: the maximum amount of data which can be read-ahead from block device. The performance of each application that is running under management OS depends on the performance of read-ahead cache. Thus, using optimal maximal read-ahead window size value for the current application can improve performance of this application significantly. Typically, the finding the optimal maximal read ahead window size for target application consists of the following steps: (1) tune operating system by current maximal read-ahead window; (2) reboot the system; (3) run the target application; (4) measure the time of execution of target application. These steps should be repeated several times for various maximal read-ahead window sizes until the optimal value of read-ahead window size will be found. The main disadvantage of such method is too much time that should be spent for finding the optimal value of maximal read-ahead window size. Thus, finding the optimal value of maximal read-ahead window size during single application run could significantly improve the performance of target application and save a lot of machine and human resources.

2 Method Overview

The suggested technique consists of the following steps: (1) user should collect data from the application and OS during application execution by monitoring tool for Linux kernel one time; (2) monitoring tool analyzes collected data and evaluates optimal maximal read-ahead window size automatically; (3) user can access evaluated optimal read-ahead window size value via monitoring tool user interface. Thus, in comparison to manual method of finding optimal read-ahead window size and other existing methods (see [11], [12], [13] for details), the suggested method has the following advantages: (1) determines optimal maximal read-ahead window size value during single run of application that is to be optimized; (2) uses existing read-ahead subsystem without any changes.

This method is designed to be used as a module of a monitoring tool such as SWAP - System-Wide Analyzer of Performance developed by Samsung Research Center (SRC) in Moscow. The initial version of SWAP was developed in 2006 and uses functional interfaces of Kprobe for providing dynamic instrumentation of Linux kernel for ARM and MIPS architecture. Next revision of SWAP tool was developed in SRC in 2007. This revision allowed collecting traces from predefined functions in Linux kernel that contains general information of system characterization. The current SWAP revision can monitor both kernel and application levels of the Linux system. It provides evaluation of a set of important system characteristics for main Linux subsystems (such as Memory Management, Process Management, File System and Network). Additional, current revision of SWAP has some automatic performance analysis features such as trace comparison, automatic bottleneck region localization, etc.

We have integrated our method into SWAP framework because SWAP satisfies to all necessary requirements of our method.

2.1 General Idea

We suggest that optimal solution can be obtained by minimization of the following two characteristics: the number of requests to the block device and the amount of pages that were forced to read by target application, but were not used.

The big value of maximal read-ahead window size provides minimization of the first characteristic – number of requests to the block device. However, in common situation the big value of maximal read-ahead window size provides reading a lot of pages from block device that are not used by application. In the other words the big value of maximal read-ahead window size makes OS performs a lot of unnecessary job that is very time consuming. It is obvious that application performance degrades because of such OS behavior.

The small value of maximal read-ahead window size provides minimization of the second characteristic – the amount of pages that were forced to read by target application, but were not used. However, in this case optimization effect of read-ahead cache is decreased because of a lot of application requests for memory pages invoke the requests to the block device instead of requests to the read-ahead cache.

Thus, to reach the optimal OS behavior for current application we need to obtain such maximal read-ahead window size value that provides minimization of both characteristics simultaneously.

2.2 Collected Data

The first step of our method requires collecting the important information about application behavior during

execution for future analyzing and evaluating the result. For these purposes, it is necessary to monitor the accesses to the memory that has been performed by application. SWAP has ability for tracking memory accesses and organizing them in trace that allows estimating the following aspects: 1) the total number of accessed memory pages, loaded from block device; 2) the addresses of accessed memory pages and their source (loaded from block device or not); 3) the order of accesses.

Notice, that our method requires the single application run only for gathering the monitoring information for evaluation of the optimal values of maximal read-ahead window size.

2.3 Analysis of Collected Data

According to the basic idea, two following values should be minimized simultaneously:

- F - number of requests to the block device;
- G - number of pages that were forced to read by target application, but were not used.

Both values - F and G - can be evaluated according to the information about memory pages that have been accessed by application.

Our method performs the emulation of read-ahead procedure for obtaining the required parameters for resolving the minimization problem. Linux 2.6 read-ahead cache has a complex behavior: after each request to the page from block device, some additional following pages are loaded into read-ahead cache. Number of additional pages is less or equal to the maximal read-ahead window size value. During emulation of read-ahead cache behavior we simplify the real OS behavior: we consider that the number of additional pages that are loaded into read-ahead cache is equal to the size of read-ahead window (see Figure 1).

This emulation procedure allows determine those pages can be potentially loaded into read-ahead cache for any predefined maximal read-ahead window size value. According to this information it is possible to evaluate the number of requests to the block device F. Thus, the number of requests to the block devices F can be described as a function:

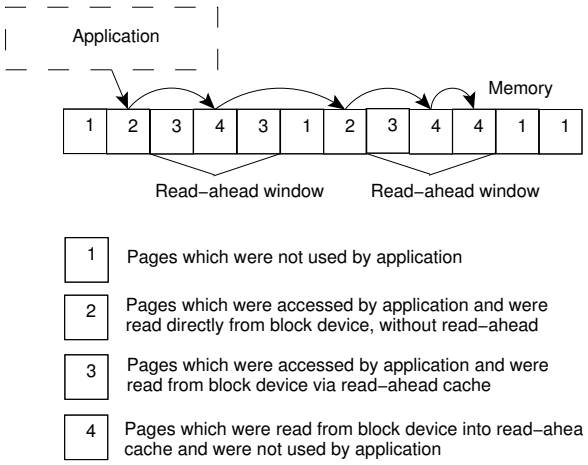

Figure 1: Read-ahead emulation procedure

$$F = Rrn(Ma, Rws) \quad (1)$$

where Rrn is the number of requests to the block device, Ma is number of memory accesses that requires mapping pages from block device and Rws is the maximal read-ahead window size.

The value of G that describes the number of pages that were forced to read by target application, but were not used, can be evaluated by taking into account the following peculiarities of Linux 2.6 read-ahead cache behavior:

- each read-ahead request produces a read operation for page from block device, which will not be placed in read-ahead cache;
- each read-ahead request additionally produces Rws read operations for pages from block device

Thus, the total amount of pages that were read from block devices within the performed read-ahead emulation can be expressed as a following function g:

$$g = Rrn * Rws + Rrn \quad (2)$$

Since the collected information on memory accesses provides information about total amount of pages requested by application, it is suitable to define the value of G (see Fig. 1) as follows:

$$G = Rrn * Rws + Rrn - Tap(Ma) \quad (3)$$

Here Tap is a total amount of pages from block device requested by application. Thus, we define G as a difference between the number of pages that have been really read from the block device as a result of the read-ahead cache behavior and the number of requested pages.

Notice that both functions F and G depend on two arguments, namely on the collected information on memory accesses (Ma) and on the read-ahead window size (Rws). The first argument Ma can be considered as a constant since the problem of performance optimization is solved for some fixed behavior of target application. Hence, both characteristics are the functions of a single argument - maximal read-ahead window size (Rws). It is possible to obtain different emulated values for both characteristics by varying Rws value.

According to the basic idea of the method the minimization problem for functions F and G should be solved. Both functions F and G reach their minimum value in extreme points satisfying to the following necessary conditions:

$$\begin{cases} \frac{\partial F}{\partial Rws} = 0 \\ \\ \frac{\partial G}{\partial Rws} = 0 \end{cases} \quad (4)$$

Since functions F and G depend on one variable Rws, we have a redefine system of two equations. To solve this incorrect problem [3], we change searching of exact solution of the redefine system of two equations by searching of its quasi-solution satisfying to solution of the following minimization problem:

$$min_{Rws}\{F^2(Rws) + G^2(Rws)\} \quad (5)$$

This minimization problem can be rewritten also as follows:

$$min_{Rws}\{Rrn^2 + [Rrn * (Rws + 1) - Tap]^2\} \quad (6)$$

Here we call an expression in figure brackets as a $coefficient of optimality$:

$$C_{opt} = \{Rrn^2 + [Rrn * (Rws + 1) - Tap]^2\} \quad (7)$$

It is easily to see that there are two limiting values for Rws which gives trivial solution exact for G:

1. $Rws = 0$, in that case each page, accessed by application is read from block device and none of the pages are read into read-ahead cache. In this case Rrn is equal to Tap. Thus, G is always equal to 0.

And for F:

2. $Rws =$ *total amount of memory in computing system*. In this case, after first access to the page with minimal address, all pages will be read into read-ahead cache.

Notice, when function F has the minimal value, then function G has the maximal value and vice versa. Both mentioned limited values of Rws allow restricting the set of possible values of Rws.

The minimization problem can be solved by the exhaustive search. However, during test period it is was noted that minimized expression is a function with one turning point - the point of minimum. We use the iterative method for speeding up the search: we emulate the read-ahead cache behavior for different values of maximal read-ahead window size Rws and choose such Rws value that makes a coefficient of optimality (7) becomes minimal.

3 Experimental Data

Using SWAP monitoring tool with our optimization method we collected several sample traces for embedded boards with running applications considered below. The appropriate results obtained in our experiments are also presented and discussed. For measuring inaccuracy of our method we use the following technique: the optimization percentage of optimal maximal read-ahead window size (found by hands) and quasi-optimal maximal read-ahead window size (proposed by our method) are measured (with comparison to default maximal read-ahead window size); the difference between optimal percentage and quasi-optimal percentage gives us the "amount" of missed optimizations. For example, the start-up of application for default $Rws = 255$ equals to 34.9 seconds; for quasi-optimal $Rws = 2$ equals to 14,5 seconds; for optimal $Rws = 1$ equals to 14 seconds. The optimization percentage of quasi-optimal Rws in comparison with default Rws is 58.45%. The optimization percentage of optimal Rws in comparison with default Rws is 59.88%. The difference between optimal and quasi-optimal solutions is $(59.88\% - 58.45\%) = 1.43\%$.

Environment	Details
Board	DTV_x260, MIPS-based
Kernel version	2.6.10
Application	Digital TV management software (exeDSP)
Initial read-ahead window size	255 pages
Performance measurement tool	SWAP
Base for memory access event information	`do_page_fault()` SWAP event

Table 1: Characteristics of DTV_x260 board and exeDSP application

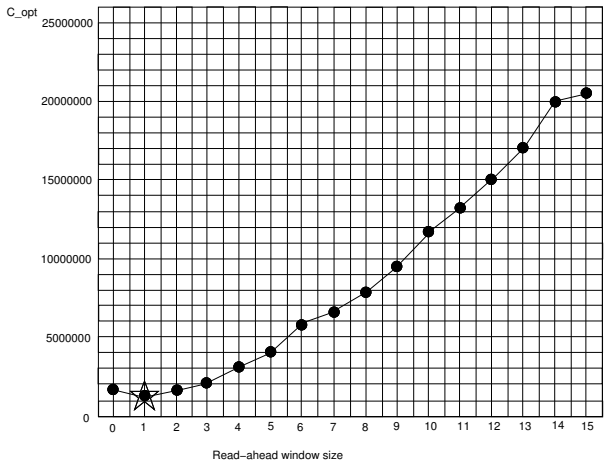

Figure 2: C_{opt} for application exeDSP. Quasi-optimal solution (point of minimum of C_{opt}) $Rws = 1$ is marked with star

3.1 Digital TV Management Application

The experiment consisted of optimizing the start-up procedure of application textitexeDSP (digital TV management software) which supervises digital TV board (DTV x260): the time of full initialization should be decreased as much as possible.

The diagnosis of our method (implemented in SWAP) was the following: quasi-optimal size of maximal read-ahead window *Rws* is equal to is 1 (see Figure 2).

The performance was measured manually for the following values of read-ahead window size enumerated in Table 2.

We can see that minimum is reached for $Rws = 1$ page

Read-ahead Window Size (Rws), pages	Time of startup, seconds
0	14.2
1	14
2	14.5
3	16
4	18.4
5	23
15	26.5
255 (default)	34.9

Table 2: Runs of exeDSP application on DTV_x260 board

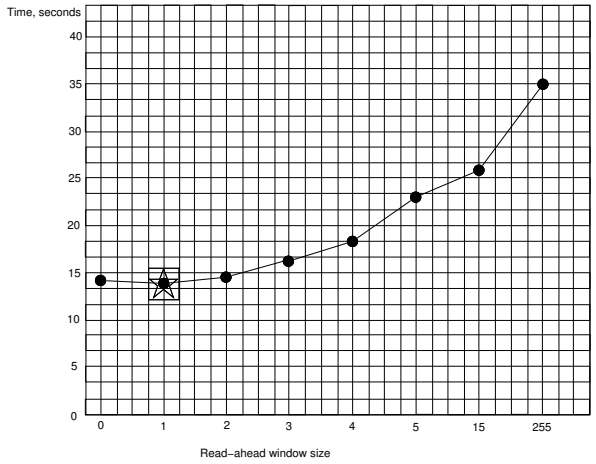

Figure 3: Start-up time for application exeDSP. Quasi-optimal solution is marked with star. Optimal solution $Rws = 1$ is marked with square

(see Figure 3). Optimal read-ahead window size is $Rws = 1$.

The difference of start-up time between default read-ahead (255 pages) and optimal read-ahead suggested by our method (1 page) is about 20.9 seconds or about 59% from default start-up time. The difference between optimal and quasi-optimal solutions is 0%, because quasi-optimal and optimal solutions are coincided.

3.2 Web Browser

The experiment consisted of optimizing the start-up procedure of FireFox (web browser) on multimedia board MVL2443: the time of full initialization should be decreased as much as possible.

Board	MVL2443, ARM based
Kernel version	2.6.16
Application	Firefox
Initial read-ahead window size	7

Table 3: Characteristics of MV2443 board and Firefox application

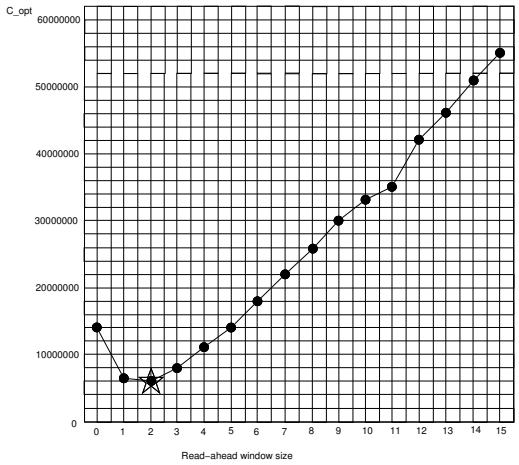

Figure 4: C_{opt} for application FireFox. Quasi-optimal solution (point of minimum of C_{opt}) $Rws = 2$ is marked with star

The diagnosis of our method (implemented in SWAP) was the following: quasi-optimal size of read-ahead window Rws is equal to 2 (see Figure 4).

The performance was measured manually for the following values of read-ahead window size enumerated in Table 4.

We can see that minimum is reached $Rws = 3$ pages (see Figure 5). Optimal read-ahead window size is $Rws = 3$. The difference of start-up time between default read-ahead (15 pages) and quasi-optimal read-ahead suggested by our method (2 pages) is about 5.8 seconds or about 7.34% from default start-up time. The difference of start-up time between default read-ahead (15 pages) and optimal read-ahead found by hands (3 pages) is about 6.3 seconds or about 7.97% from default start-up time. The difference between optimal and quasi-optimal solutions is $(7.97\% - 7.34\%) = 0.65\%$.

3.3 MP3 player

The experiment consisted of optimizing the file loading procedure of MadPlay MP3 player on the board

Read-ahead Window Size (Rws), pages	Time of startup, seconds
0	75
1	73.1
2	73.2
3	72.7
4	73.5
5	74.5
7	79
15 (default)	79
31	81

Table 4: Runs of Firefox application on MV2443 board

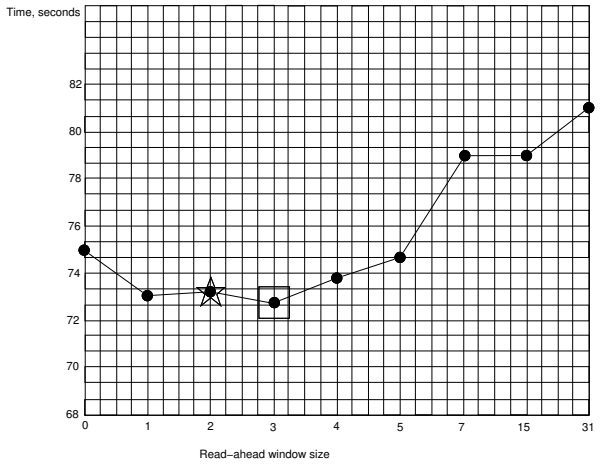

Figure 5: Start-up time for application FireFox. Quasi-optimal solution is marked with star. Optimal solution $Rws = 3$ is marked with square

OMAP5912OSK: the time of playback with null output device should be decreased as much as possible.

The diagnosis of our method (implemented in SWAP) was the following: quasi-optimal size of read-ahead window Rws is equal to 2 (see Figure 6).

The performance was measured manually for the following values of read-ahead window size enumerated in Table 6.

We can see that minimum is reached for value $Rws = 3$ (see Figure 7). Optimal read-ahead window size is $Rws = 3$. The difference of start-up time between default read-ahead (15 pages) and quasi-optimal read-ahead suggested by our method (2 pages) is 0.5 seconds or about 4.70% from default start-up time. The difference of start-up time between default read-ahead (15

Board	OMAP2912OSK ARM based
Kernel version	2.6.10
Application	MP3 Player (Madplay)
Initial read-ahead window size	15

Table 5: Characteristics of OMAP5912 OSK board and MadPLay application

Read-ahead Window Size (Rws), pages	Time of startup, seconds
0	10.66
2	10.13
3	10.09
4	10.17
5	10.17
6	10.23
7	10.31
8	10.31
15 (default)	10.63
19	10.73
31	10.95

Table 6: Runs of MadPLay application on OMAP5912 OSK board

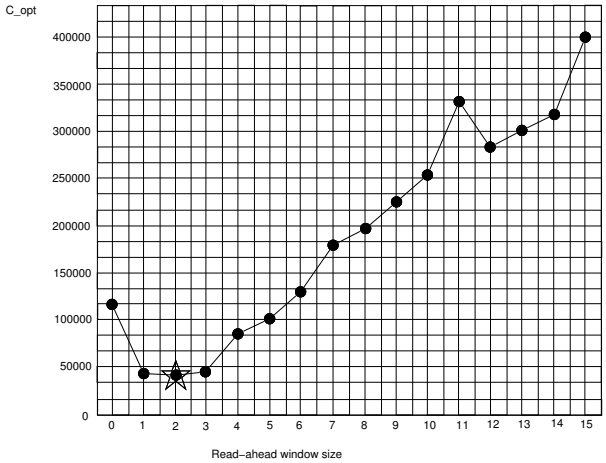

Figure 6: C_{opt} for application MadPlay. Quasi-optimal solution (point of minimum of C_{opt}) $Rws = 2$ is marked with star

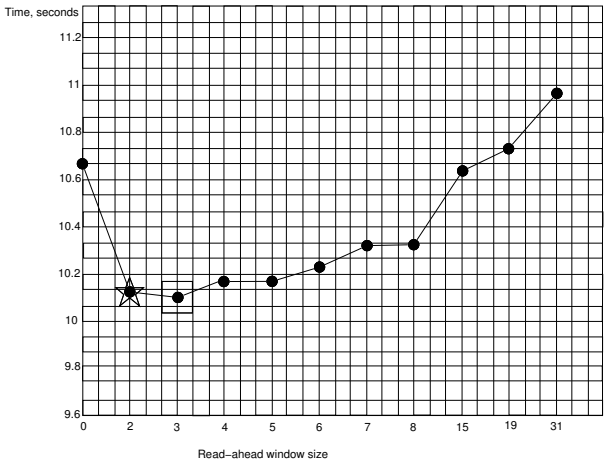

Figure 7: Time of input file loading for application MadPlay. Quasi-optimal solution is marked with star. Optimal solution $Rws = 3$ is marked with square

pages) and optimal read-ahead found by hands (3 pages) is 0.54 seconds or about 5.07% from default start-up time. The difference between optimal and quasi-optimal solutions is $(5.07\% - 4.70\%) = 0.37\%$.

4 Conclusion

We have presented a new method for automatic evaluation of optimal value for single Linux tunable parameter: maximal read-ahead window size. Tuning of this Linux parameter by obtained value can improve performance of embedded application significantly. Our method is simple to use and requires single run of application for evaluation of all necessary characteristics. It does not

require any additional knowledge in system or optimization areas and can be used by any application developer who wants to increase performance of his application by reducing hidden performance bottlenecks in Linux operating system. Due to its advantageous characteristics, this method can be widely used for optimization of embedded systems to increase their quality and effectiveness.

References

[1] Daniel P. Bovet, Marco Cesati, Understanding the Linux Kernel, Third Edition, O'Reilly Media, 2006.

[2] Linux kernel source files (`http://kernel.org`).

[3] V. A. Morozov, Regular Solution Methods of Non-Correct problems, Nauka, Moscow, 1987.

[4] James R. Larus and Eric Schnarr, Computer Sciences Department, University of Wisconsin, Madison, EEL: Machine-Independent Executable Editing. (`ftp://ftp.cs.wisc.edu/wwt/pldi95_eel.ps`).

[5] Amitabh Srivastava, Alan Eustace, ATOM: A System for Building Customized Program Analysis Tools.

[6] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, and Brian Bershad, University of Washington, Brad Chen, Harvard University, Instrumentation and Optimization of Win32/Intel Executables Using Etch.(`http://etch.cs.washington.edu/etch-usenixnt/etch-usenixnt.html`).

[7] Ananth Mavinakayanahalli, Prasanna Panchamukhi, Jim Keniston, Anil Keshavamurthy, Masami Hiramatsu, Probing the guts of Kprobes, Ottawa Linux Symposium, pp.101-114, July 2006.

[8] Frank Ch. Eigler, RedHat, Problem Solving With Systemtap. (`http://sourceware.org/systemtap/wiki/OLS2006Talks?action=AttachFile&do=get&target=problem-solving-with-stap.pdf`).

[9] David J. Pearce, Paul H.J. Kelly, Tony Field, Uli Harder, Imperial College of Science, Technology and Medicine, London, GILK: A dynamic instrumentation tool for the Linux Kernel.

[10] Giridhar Ravipati, Andrew R. Bernat, Nate Rosenblum, Barton P. Miller and Jeffrey K. Hollingsworth, Toward the Deconstruction of Dyninst, Technical Report, Computer Sciences Department, University of Wisconsin, Madison (`ftp://ftp.cs.wisc.edu/paradyn/papers/Ravipati07/SystemtabAPI.pdf`).

[11] US5809560, Adaptive read-ahead disk cache.

[12] US2005154825A1, Adaptive file read-ahead based on multiple factors.

[13] US2003115410A1, Method and apparatus for improving file system response time.

Proceedings of the Linux Symposium

July 13th–16th, 2010
Ottawa, Ontario
Canada

Conference Organizers

Programme Committee

Proceedings Committee