Scheduler-Assisted Prefetching: Efficient Demand Paging for Embedded Systems

Stanislav A. Belogolov[1], Jiyong Park[1], Jungkeun Park[2] and Seongsoo Hong[1]

[1]*Real-Time Operating Systems Laboratory*
Seoul National University, Seoul, Korea
{stas, parkjy, sshong}@redwood.snu.ac.kr

[2]*Dept. of Aerospace Information Engineering*
Konkuk University, Seoul, Korea
parkjk@konkuk.ac.kr

Abstract

Embedded systems tend to use demand paging in order to provide more memory to applications in a cost-effective manner. However, demand paging drastically degrades the performance when the page fault rate is high. Prefetching has been known as a common remedy for page fault overhead. Although many prefetching mechanisms have been proposed, they are either effective only for specific page access patterns or too straight-forward to decrease a page fault rate to an acceptable level. We propose a scheduler-assisted prefetching mechanism which does not have such fundamental defects. As a proof of concept, our mechanism was completely implemented in Linux. We have also conducted a series of experiments to show its effectiveness. The experimental results showed a significant improvement: the number of the major page faults and the scheduling latency decreased by 30% and 51%, respectively.

1. Introduction

Demand paging is becoming widely adopted in embedded systems as embedded applications get larger in memory footprint. Typical embedded systems used to store code in NOR flash memory because it supports eXecute-In-Place (XIP). However, an increasing number of embedded systems come with only NAND flash memory because they become subject to tighter memory budget constraints and stronger limitations on a form factor. Unfortunately, NAND flash memory cannot support XIP since NAND flash memory allows only block data transfers. This forces real-time operating systems to use demand paging to dynamically load code pages from a NAND flash memory-based storage device.

Demand paging, however, introduces unacceptable run-time overheads caused by page faults. Page fault handling incurs slow I/O data transfers from NAND flash memory and suspends the faulted task until the required page is properly loaded into RAM. As a result, the intensive demand paging drastically decreases applications throughput and response time.

Obviously, a real-time operating system could improve applications performance if it could load the required pages before they cause page faults. This is a widely practised technique known as prefetching. The important issues of prefetching are to decide which pages to prefetch and when.

This paper introduces a scheduler-assisted prefetching mechanism as a general solution to the page fault overhead problem without limitations on hardware and task sets. Our approach is to modify a scheduler to predict the task sequence and modify the virtual memory manager so that it accumulates recently faulted pages by monitoring page faults. For each task, we have its accumulated pages prefetched before it gets scheduled for execution.

As a proof of concept, we implemented our scheduler-assisted prefetching into the Linux kernel and conducted a series of experiments. The experiments showed significant performance improvement in terms of page faults and interactivity.

There have been many prefetching mechanisms in the literature. The naïve one-block look ahead prefetching [13] is certainly useful for sequential access patterns but does not work well for non-linear ones. Other works [10, 14, 15, 16, 17] focus on data cache prefetching in L2 by detecting complex data access patterns. These approaches have limited applicability because they are hardware-specific. The mechanism suggested by Lin, et al. [12] substitutes LRU-related caching by prediction-based prefetching. Since such prediction is made possible in a limited and predefined set of applications with known execution traces, the approach can be applied only for a limited set of embedded systems.

In 2001, Suh, et al. [1] and Chiou, et al. [2] suggested to use a scheduler to predict a job sequence and prefetch jobs just before they are scheduled for execution. Their work showed the theoretical decrease in the number of page faults. Using simulation, they also evaluated if it would be practical to develop such a prefetching mechanism. However, they did not present a complete prefetching mechanism.

It is also worth clarifying the difference between scheduler-assisted prefetching and two recently created prefetching mechanisms: SuperFetch [19] implemented in Microsoft Widows Vista and Swap Prefetch [18] implemented in -ck patches of Linux. Their main goal is to resolve so called "after lunch syndrome." When one leaves working computer idle for some time during lunch time and then finds it with all the tasks swapped out by the background services like an antivirus or a search indexer. That is why when one begins working again the computer starts an intensive swapping in. SuperFetch and Swap Prefetch are idle time prefetching techniques. Unlike them, the scheduler-assisted prefetching attempts to reduce the page fault rate when the system is under heavy memory load and several tasks are competing for the CPU simultaneously.

The rest of this paper is organised as follows. Section 2 introduces the main problems of the prefetching technique in a formal way. Section 3 explains the scheduler-assisted prefetching mechanism in the context of an abstract operating system. Section 4 is devoted to our implementation of the proposed mechanism on top of the Linux kernel. Section 5 provides a detailed description of the experimental set-ups, goals, benchmarking tools and results. Finally, Section 6 summarizes and concludes this paper.

2 Problem Definition

Our general goal is to decrease the number of page faults by prefetching pages which are currently absent from memory and will be accessed after the next context switch. We should address the following four issues in order to make the prefetching efficient:

1. The correct moment for starting the prefetching during the current job execution has to be determined.
2. The task that will be executed after the next context switch has to be determined.
3. The pages that will be accessed by the predicted task have to be determined.
4. The pages in RAM that can be replaced by prefetched pages without degradation of the current task performance have to be determined.

In this section, we formally define a problem for each of the issues to formulate our scheduler-assisted prefetching mechanism.

2.1 Activation Moment Problem

The first problem is to determine the correct moment for starting the prefetching. We call this moment – an activation moment. We formulate the problem with a boolean function defined as below.

$$\mathrm{A}(t)=\begin{cases}1 & \text{, if the prefetching is required at time } t\\ 0 & \text{, otherwise}\end{cases}$$

This function tells us whether the current time is the right moment for starting the prefetching or not. Prefetching is activated only when the function returns 1. The function should be defined in a way that the impact on the performance is minimized. An early prefetching may degrade the performance of the currently running task by swapping out or discarding pages that are currently used by the task. A late prefetching can degrade the performance of the next task since there is not enough time for loading pages into RAM.

2.2 Task Prediction Problem

Given a correct activation moment, the task that will be executed after the next context switch must be determined. Let t be a given activation moment that satisfies $A(t)=1$. Also, let T_t denote a set of ready tasks at activation moment t. Then we can predict next task $\tau \in T_t$ using a scheduler function as below.

$$\tau = \mathrm{S}(T_t)$$

This function returns the task to be executed right after the next context switch among the ready tasks according to the given scheduling algorithm.

2.3 Page Selection Problem

Given selected task τ predicted at activation moment t, the next problem is to determine the pages to be prefetched for task τ. This problem can be formulated as a function defined below.

$$\mathrm{P}(\tau, t)=(p_1, p_2, \ldots)$$

This function gives a sequence of pages that will be accessed by task τ.

2.4 Page Replacement Problem

The last remaining problem is to determine the pages that are replaced by the prefetched pages. Usually, an operating system has a default page replacement policy which is used by its implementation of the demand paging. However, prefetching changes the normal page access patterns. Therefore, we need to either ensure that our prefetching does not cause harmful effect to the default page replacement policy or suggest our own page replacement policy. In other words, we are required to answer the following questions:

1. How does the typical page replacement policy behave when our prefetching mechanism is enabled?
2. Can our prefetching mechanism cause performance degradation if we use the default

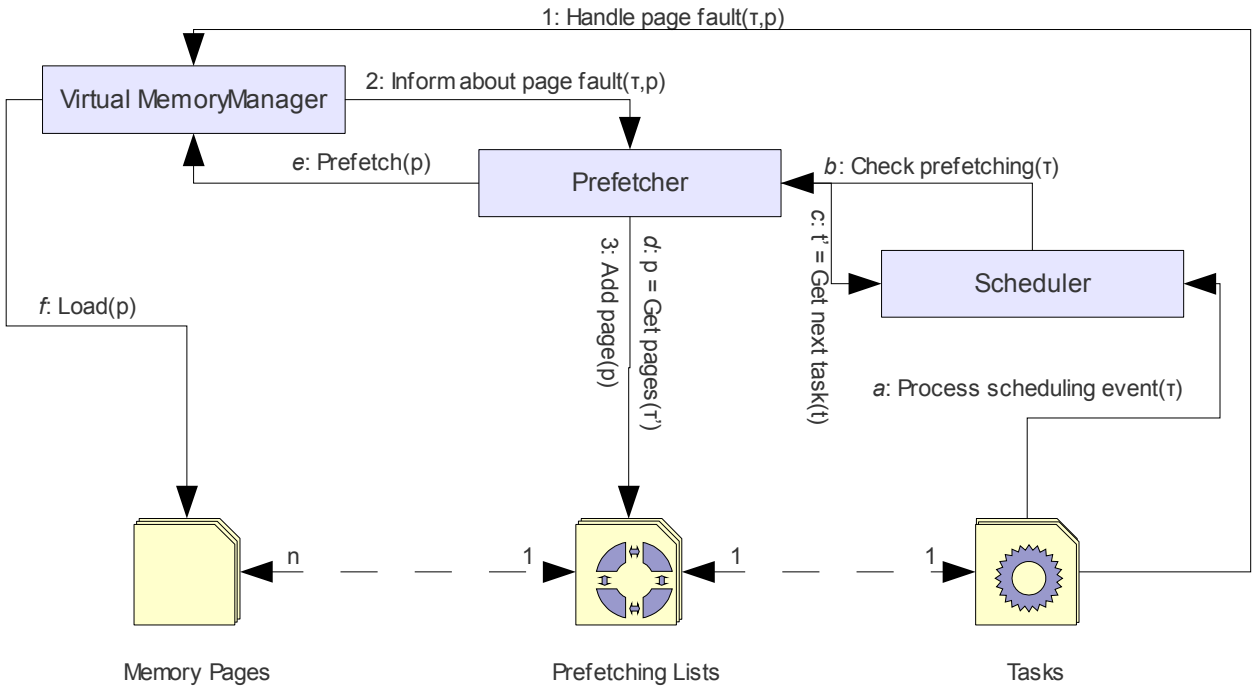

Figure 1: Scheduler-Assisted Prefetching Mechanism.

replacement policy?

By answering the first question, we analyse the impact of our prefetching mechanism on the default page replacement policy. The negative answer on the second question ensures no performance degradation. However if the answer on the second question is positive, then we have to change the default page replacement policy in order to avoid such regression.

3 Scheduler-Assisted Prefetching Mechanism

This section introduces the scheduler-assisted prefetching mechanism. The main idea is that the memory pages of a next scheduled task are loaded before the context switch in parallel with execution of a currently scheduled task. This requires computing systems to be able to load data from the storage device in parallel with another task execution. This is true for the most computing systems because they usually have integrated DMA controllers.

Figure 1 shows general overview of the scheduler-assisted prefetching mechanism. This Figure illustrates two activities which make up the scheduler-assisted prefetching: accumulation of faulted pages references and loading of pages used by the task, which is scheduled next. The accumulation activity is numbered by Arabic numbers starting from 1 and the page loading activity is numbered by lower case English letters starting from '*a*'. Also this Figure shows the multiplicity relationship between memory pages, prefetching lists and tasks. Each task has one prefetching list, which references multiple memory pages.

As it was mentioned in the previous section, we have four issues to design the prefetching system. These issues are covered in the next subsections.

3.1 Activation Moment Calculation

The first issue we discuss is the moment when the prefetching procedure is executed. Basically, we have two approaches for prefetching activation:

1. Timer-based approach suggests setting up the prefetching function as a handler for a timer. We set up such a timer during rescheduling. This timer will interrupt a current task some time before it finishes its job to give the prefetcher a chance to start loading pages for the next task.
2. Scheduler tick-based approach suggests to decide prefetching initiation during a scheduler tick. Usually, during a scheduler tick we can calculate how much time left before the rescheduling execution. Using this information we decide if it is time to start prefetching.

The fundamental difference of these approaches is that the first one calculates prefetching starting time relative to a left border of the task execution period – the moment when a job is released, while the last approach puts that moment relative to a right border – the moment when a task is suspended.

The disadvantage of the first method is that we need

to do complex and scheduler-dependant calculations of the waiting time for each task. These calculations should take into account priority system of an OS. For example, Linux kernel has 40 priorities and a time unit of execution is different not only for different priorities, but also for the same priorities of different schedulers and even for different versions of the same scheduler. The second method does not have this disadvantage, but assumes that scheduler is implemented using scheduling ticks. In our implementation we used the tick-based approach.

Now we will discuss how we calculate the right moment for the prefetching activation. Let us introduce variable parameter $\xi \in [0, 1]$ which determines what fraction of a current task should be completed before we start prefetching for the next task. Smaller values of ξ corresponds to earlier prefetching, while $\xi = 1$ means that no prefetching is required. We can calculate each moment of prefetcher activation using formula defined below.

$$t = r_{cur} + \xi (r_{S(t)} - r_j)$$

In this formula r_{cur} is the release time of a current task and $r_{S(t)}$ is the release time of the predicted task.

3.2 Task Sequence Prediction

In order to do prefetching as the Figure 2 shows, we need to know the task which will be executed next after a current task.

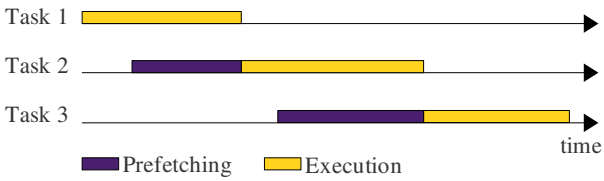

Figure 2: Next task prefetching.

We modify the task scheduler, so that it calls original scheduler two times to provide us with a task which is predicted for execution next as it shown by activities *a*, *b* and *c*. Predicted means that this task will be rescheduled for execution unless some more important task appears in a run-queue.

Usually, an operation of selecting the next task for execution is as simple as selecting first element from a certain collection. Every scheduling algorithm tries to optimise such operation in order to decrease context switch overhead, so this operation is not expensive. Our approach suggests that this operation should be called twice during one job execution: once for deciding which task to prefetch and once to select next task during context switch. When prefetcher is activated, it makes all the pages, referenced in corresponding prefetching list, present in memory as activities *d*, *e* and *f* show.

Of course, at the rescheduling moment scheduler may return a task which is different form the task selected for prefetching because a new task has been started or a temporally suspended high priority task was waken up. In that case prefetching will not give any performance boost, but will not degrade it as well for the reasons discussed in Section 3.4.

In order to make performance even higher, we can further modify task scheduler so that it always tries to follow its earlier prediction. This ensures almost 100% effectiveness of prefetching for the price of a less responsive scheduler. We believe such approach can be useful for the soft real-time and general-purpose operating systems.

3.3 Prefetching Pages Accumulation

Trying to prefetch all pages of a given task is impractical because the sum of the memory footprints of a current and next task can exceed RAM size. Also applications with large memory footprints usually actively use only limited number of pages.

We know that for every task the sets of pages used during its current job and during its next job executions differ insignificantly [20]. Usually, the working set changes slowly and gradually as Figure 3 shows.

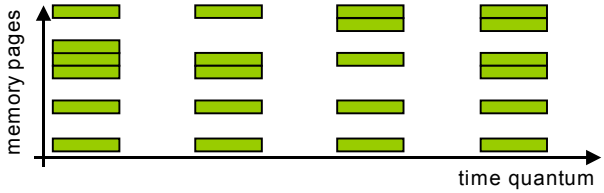

Figure 3: Page access pattern.

We accumulate a list of recently faulted pages in order to avoid them in the future as shown by activities 1 and 2 in Figure 1. The references to these pages are stored in a list associated with memory descriptor of each task. The reference to a page is added to the list when the page is being faulted as represented by activity 3.

3.4 Replacement Policy Analysis

In this subsection we assume that a task set is big enough to consume all available memory and every request for a new memory page initiates swapping out or discards a code page. Such conditions we will call a *heavy memory load*.

The Least Recently Used (LRU) algorithm is known to be the best page replacement algorithm. The idea of LRU algorithm is that an operating system maintains a list of all memory pages sorted by the last access time in descending order. When a new page addition is required, the last page in the list is swapped out or discarded, a

new page is allocated and added to the head of the list. Since a plain implementation of this algorithm is impractical, modern operating systems typically use algorithms which mimic the LRU behaviour exploiting different kinds of heuristics. That is why we will consider the LRU algorithm as a memory replacement policy.

Analysing the LRU algorithm behaviour under a heavy memory load, we found a very negative implication which drastically degrades performance. The problem is that all the pages of a given task are swapped out or discarded just before this task execution is resumed because at that moment these pages are actually **least recently used**. Figure 4 illustrates such situation. This example considers a system with 4 tasks and RAM large enough for pages of only 3 tasks. Tasks are run in the Round-Robin manner. The pages of each task come to the beginning of the LRU list during this task execution and then migrate towards end of the list as other tasks are executed. Notice that the pages of a task are completely forced out by the release time of its next job.

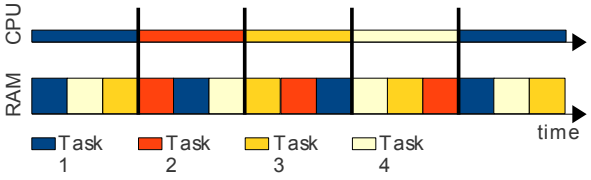

Figure 4: LRU under heavy memory load.

The extreme case of this situation, which is called thrashing, is when a system progresses very slowly because it is constantly handling page faults. Reference [5] provides a detailed discussion of this problem and suggests a workaround which eventually was implemented in the Linux kernel.

The scheduler-assisted prefetcher addresses this problem. Since it makes the pages, which are likely to be accessed, be present in memory, these pages automatically move to the beginning of the LRU list. The more accurate approach is to artificially put prefetched pages in the LRU list after the pages which were added by the current task during its current time unit. In that case we avoid situation when prefetched pages force out pages of the current task, thus we will guarantee no performance degradation. Figure 5 illustrates the LRU page list behaviour when prefetching is on. Notice that the scheduler-assisted prefetcher puts pages of the next task after the pages of the current task.

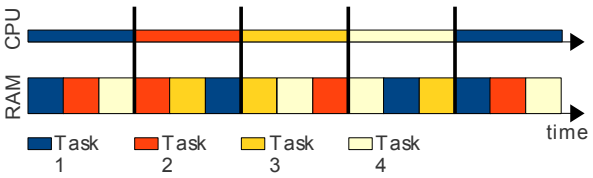

Figure 5: LRU behaviour with prefetching.

4 Implementation

The proposed mechanism was implemented in Linux kernel version 2.6.23-rc2. In this section we discuss Linux- and implementation-related issues of prefetching. Since the Linux kernel is written in C, we provide our algorithms in C. In order to avoid insignificant details of the Linux API, we will use a simplified pseudo version of this API. The implementation of the scheduler-assisted prefetching required modification in two places of the kernel:

1. Process Scheduler – *sched_fair.c*;
2. Virtual Memory – *mm_types.h, memory.c*;

This kernel uses a new scheduler – CFS (Completely Fair Scheduler [11]). The prefetching was implemented on top of the CFS. During every scheduling tick we check which part of the time unit the current task has already used. If it has used more than ξ of the time unit, we find the next task and start prefetching for it as shown in algorithm 1. The prefetching routine forces loading of recently swapped out or discarded pages of the next task. Algorithm 1 uses the following attributes and methods:

- `rq->nr_tasks` – number of tasks in run queue;
- `rq->curr` – currently executing task;
- `rq->curr->ex_part` – part of time unit already consumed by current task;
- `rq->get_next_to(x)` – returns task next to `x` in run queue;
- `next->prefetched` – prefetching flag, which states if prefetching for the task has already been executed.

```
Algorithm 1: Prefetching part of the
scheduling tick routine.
Input:
      pref_start - prefetching start
value rq - run queue
Output: none
if(rq->nr_tasks > 2 &&
   rq->curr->ex_part > pref_start){
   next = rq->get_next_to(rq->curr);
   if(next->prefetched)
      return;
   forall(p in next->prefetch_pages)
      make_present(p);
   next->prefetched = true;
```

In order to store and manage a list of recently faulted pages, we had to modify memory map structure and routine responsible for its creation. These changes allow us to accumulate a list of pages, which the prefetching routine will try to make present later as shown in algorithm 2. Algorithm 2 uses the following attributes and methods:

- `handle_fault()` – page fault handler, returns page fault type;
- `curr->prefetch_pages->add` – adding operation of the prefetching list;
- `count_vm_event()` – virtual memory events counting operation;
- `PF_MAJOR` – major page fault flag.

```
Algorithm 2: Prefetching part of the page
fault routine.
Input:
      fp – faulted page
      curr – currently executing task
Output: none
if(handle_fault(fp, curr) == PF_MAJOR){
   curr->prefetch_pages->add(fp);
   count_vm_event(PF_MAJOR);
}
```

Linux distinguishes several kinds of page faults due to several possible reasons they are initiated and ways they are handled. The page faults which induce I/O operations are called *major*. The major page faults are the longest to process, since they require slow I/O operations. In contrast to major page faults, all other page faults take little time to handle. However, they still consume some processor time. We count page faults because we will need this number during experiments.

Page Fault is not an exceptional situation in the Linux kernel as it may seem. In fact, it takes more than 700 000 of them just to boot up our testing machine. Such situation exists because Linux tends to use lazy algorithms for its virtual memory management. Normally, a number of the major page faults is a very small fraction of total number of page faults.

5 Experiments

The goal of our experiments was to show that prefetching gives actual performance boost. The expected result was that the number of page faults would drop. However, this fact alone does not guarantee performance improvement because the prefetching overhead can be too large. In order to check that prefetching overhead does not degrade performance, we performed the interactivity tests.

5.1 Benchmarking tools

We used two tools for the experiments called Stress [8] and Interbench [9]. The first program was used to emulate heavy memory load situation and the second – to evaluate the impact on the interactivity.

We designed the Stress test in a way to simulate the situation showed by the Figure 4. Performing the stress tests, we were interested in the number of major page faults and the overall number of page faults.

We used Interbench for the interactivity tests. This benchmark was designed to emulate the CPU scheduling behaviour of the interactive tasks. We can measure the impact of prefetching on interactivity by comparing a scheduling latency and amount of desired CPU time tasks receive. The scheduling latency represents the time from the sleep till the task gets scheduled.

5.2 Experimental Set-ups

The experiments were performed on emulated machine of the following configuration:

- Single x86 CPU
- 32 or 64 MB of RAM
- 256 MB of Swap
- Gentoo Linux Minimal

An emulator process was executed with the highest priority during every test to ensure no influence from the host system. The prefetching list was limited with 1024 page references.

We used three kinds of the Linux kernel for testing:

1. A vanilla kernel[1] version 2.6.24-rc2.
2. The same kernel patched for prefetching after 75% of a time slice is used. We call it prefetch-75.
3. The same kernel patched for prefetching after 50% of a time slice is used. We call it prefetch-50.

During the stress test 5 virtual memory workers were forked. Each worker allocates 8 MB and accesses allocated memory periodically. Size of each page is 4 KB, thus prefetcher can prefetch up to 50% of memory allocated by each task. The total amount of required memory is 40 MB and it exceeds the amount of available RAM on 32 MB machine. Since Linux mimics LRU behaviour for the page replacement policy, each worker is completely forced out from the memory before its next iteration. We also conducted this experiment with 64 MB of RAM to evaluate the effect of prefetching when a system has enough memory for a given task set.

For the interactivity test we used the *gaming* work simulation with the *memload* background. The *gaming* work simulation corresponds to 100% CPU utilization, when the *memload* corresponds to a heavy memory and swap pressure by repeatedly accessing 110% of available ram and moving it around and freeing. Reference [9] provides complete description of Interbench work

1 Vanilla kernel – the Linux source tree released by Linus Torvalds without any other modifications.

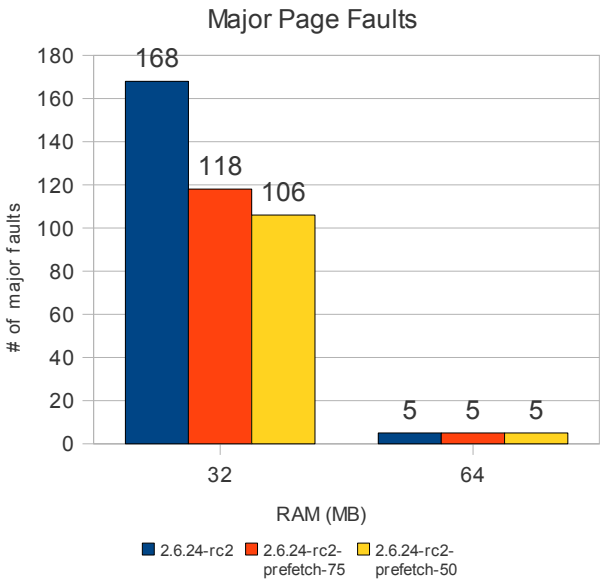

Figure 6: Number of major page faults.

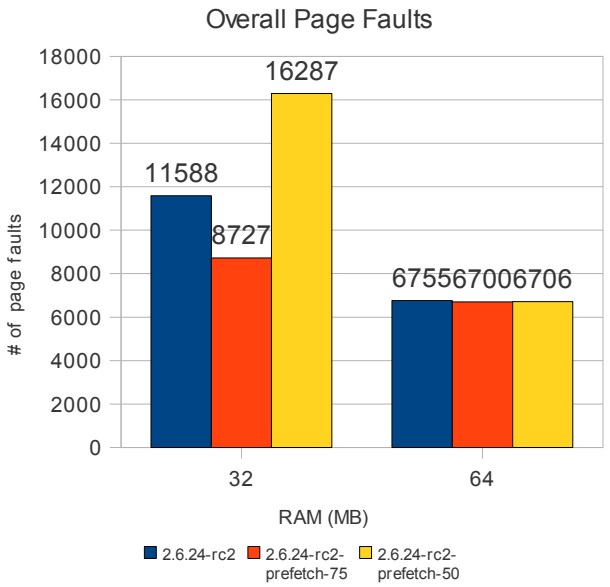

Figure 7: Overall number of page faults.

simulations and loads.

5.3 Experimental Results

The results of the Stress test are showed by Figures 6 and 7. The Figure 6 shows that both prefetchers significantly reduce the number of the major page faults. The prefetch-50 reduces number of the major page faults stronger. This result is predictable because the earlier we start prefetching pages, the more pages we managed to load into memory before a context switch happens. Obviously, starting prefetching just after a context switch would give maximum effect on the number of the major page faults.

However, starting too early negatively influence the overall number of page faults. In case of prefetch-50, the overall number of page faults has increased by 40% and is almost twice as large comparing to prefetch-75. Figure 7 shows this negative effect. Of course, non-major page faults require much less handling time, but still they interrupt execution of a task. It also worth noticing that memory reduce from 64 MB to 32 MB results in only 29% increase of overall page fault for prefetch-50, while without prefetching this number is 72%.

Table 1 summarises the impact of prefetching during the stress test. As one can see, earlier prefetching has a strong negative impact on the total number of page faults when the prefetching done at the right time can significantly decrease both number of the major page faults and the total number of page faults.

Table 1: Change of PF rate after prefetching.

Kernel	Mjr. PF Change (%)	Total PF Change (%)
Prefetch-75	-30	-25
Prefetch-50	-37	+40

The interactivity tests showed noticeable reduction in both average and standard deviation values of the scheduling latency. Smaller scheduling latency means smoother playback of multimedia and better responsiveness on user actions. Also the percentage of time unit used by tasks has increased from 61.3% to 76.5% for prefetch-75 and 74.4% for prefetch-50. The table 2 summarises the impact of prefetching during the interactivity test.

Table 2: Interactivity benchmark results.

Measurement	No prefetching	Prefetch-75	Prefetch-50
Avg. Latency (ms)	63.1	30.7	34.4
Latency SD (ms)	90.1	60.3	61.1
% of Desired CPU	61.3	76.5	74.4

During our tests, we did not try to find an optimal moment for prefetching because this value heavily depends on a task set, memory size and storage device throughput. We are going to address this issue in our future work.

6 Conclusions

In this paper we discussed a problem of performance degradation because of a high page fault rate. The main reasons is a page fault handling overhead. When the number of page faults is high, so is the handling overhead.

The suggested solution of the problem is to use the scheduler-assisted prefetching. The decision which pages to prefetch is made with the assistance of a task scheduler and page faults monitoring. We suggest a scheme that can be used for implementation of our prefetching for a given operating system. This scheme deals with the four main problems of prefetching.

First, we address the activation moment problem by choosing a fraction of time unit after the execution of which the prefetching should be activated. The task prediction problem can be addressed by extra call of scheduler at prefetching moment. As a solution of the page choosing problem, we suggest to accumulate limited list of last faulted pages for each task. Finally, analysis of the page replacement problem showed that LRU with few modifications can be applied.

Our solution follows "do no harm" and "best effort" approaches. This means that our prefetching can not guarantee zero number of page faults but can avoid performance degradation. The implication is that our scheduler-assisted prefetching does not guarantee improvement of schedulability but will not hurt it as well. This is an important issue for the real-time systems.

In order to measure performance improvement, we have implemented prefetching on top of the Completely Fair Scheduler of Linux and have performed experiments. We conducted two kinds of experiments: Stress test of the virtual memory and Interactivity experiment. The Stress experiment showed a noticeable drop of page faults numbers: 30% for major page faults together with 25% for overall page faults. The Interactivity experiment showed the following improvement of the system interactivity: the scheduling latency average decreased by 51%, the standard deviation – 33%. Thus, our experiments confirm feasibility of the scheduler-assisted prefetching.

There are two issues we would like to address in our future work. First, we want to develop a formal theory of our prefetching for a real-time scheduling, so that it could be used by the hard real-time systems. If the scheduling is absolutely predictable and the memory footprints of tasks are known, then our mechanism can decrease WCET and thus enhance schedulability. Second, we are intended to develop an algorithm for smooth on-line variation of the activation moment depending on the memory size and utilization.

References

[1] Ed Suh, et al., "Job-Speculative Prefeetching: Eliminating Page Faults From Context Switches in Time-Shared Systems," *MIT Computation Structures Group Memo 442,* 2001.

[2] Derek Chiou, et al., "Scheduler-Based Prefetching for Multilevel Memories," *MIT Computation Structures Group Memo 444,* 2001.

[3] Mel Gorman, *Understanding the Linux Virtual Memory Manager,* Prentice Hall PTR, 2004.

[4] Daniel P. Bovet, Marco Cesati, *Understanding the Linux Kernel*, 3rd Edition, O'Reilly, 2005.

[5] Song Jianga, Xiaodong Zhangb, "Token-ordered LRU: an effective page replacement policy and its implementation in Linux systems," *Performance Evaluation*, 60, 5–29, 2005.

[6] Cao, P. Felten, E. W. Karlin, A. R. Li, K., "A Study of Integrated Prefetching and Caching Strategies," *Performance Evaluation Review*, Vol. 23, No. 1, 1995.

[7] Patterson, R. H. Gibson, G. A. Ginting, E. Stodolsky, D. Zelenka, J., "Informed Prefetching and Caching," *Operating Systems Review*, Vol. 29, No. 5, 1995.

[8] Amos Waterland, Stress benchmarking tool, http://weather.ou.edu/~apw/projects/stress/

[9] Con Kolivas, Interbench, http://members.optusnet.com.au/ckolivas/interbench/

[10] Ronald G. Dreslinsky, Ali G. Saidi, Trevor Mudge, Steven K. Reinhardt, "Analysis of Hardware Prefetching Across Virtual Page Boundaries," *CF'07*, May 7-9, 2007.

[11] Ingo Molnar, "Completely Fair Scheduler Design," http://people.redhat.com/mingo/cfs-scheduler/sched-design-CFS.txt

[12] Jian-Hong Lin, Yuan-Hao Chang, Jen-Wei Hsieh, Tei-Wei Kuo, Cheng-Chih Yang, "A NOR Emulation Strategy over NAND Flash Memory," *RTCSA'07*, August 21-24, 2007.

[13] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," *In Proc. 17th Ann. Int'l Symp. on Computer Architecture*, pages 364–373, 1990.

[14] J. W. C. Fu, J. H. Patel, B. L. Janssens, "Stride directed prefetching in scalar processors," *25th Ann. Int'l Symp. on Microarchitecture*, pages 102–110, 1992.

[15] S. Kim, A. V. Veidenbaum, "Stride-directed prefetching for secondary caches," *International Conference on Parallel Processing*, pages 314–323, 1997.

[16] K. J. Nesbit, A. S. Dhodapkar, J. E. Smith, "Ac/dc: An adaptive data cache prefetcher," *Proc. 13th Ann. Int'l Conf. on Parallel Architectures and Compilation*

Techniques, pages 135–145, 2004.

[17] K. J. Nesbit, J. E. Smith “Data cache prefetching using a global history buffer,” *Proc. 10th Int’l Symp. on High-Performance Computer Architecture (HPCA)*, page 96, 2004.

[18] Con Kolivas, “Swap Prefetch,” *Linux kernel mailing list* <linux-kernel@vger.kernel.org>.

[19] Mark Russinovich, “Inside the Windows Vista Kernel: Part 2,” *TechNet Magazine*, March, 2007.

[20] Abraham Silberschatz, Peter Galvin, Greg Gagne, *Applied Operating Systems Concept*, John Wiley & Sonsm Inc., pages 325–328, 2000.