Scheduler-Assisted Prefetching: Efficient Demand Paging for Embedded Systems

Stanislav A. Belogolov, Jiyong Park and Seongsoo Hong
Real-Time Operating Systems Laboratory
Seoul National University, Seoul, Korea
{stas, parkjy, sshong}@redwood.snu.ac.kr

Abstract

In an attempt to substitute NOR flash with NAND flash and provide more memory to applications, embedded systems have to use demand paging. However, demand paging drastically degrades performance when the page faults rate is high. Prefetching has been known as a common remedy for the page fault overhead. Although many prefetching mechanisms have been proposed, they are efficient only for specific page access patterns. Others tend to be either too naïve to provide impressive results or too difficult to implement. We propose the scheduler-assisted prefetching mechanism which does not have such fundamental defects. As a proof of concept, this mechanism was completely implemented in Linux. We also have conducted a series of experiments to show the effectiveness of our approach. 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

Usually, embedded systems have tight memory budgets and strong limitations of a device size. That is why an increasing demand for memory created by embedded systems applications forces producers to use demand paging. However, the demand paging makes an application wait until page fault handling operations are completed. As a result, the intensive demand paging drastically decreases applications throughput or 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 most important problems are to decide which pages should be prefetched and when to prefetch.

This paper introduces a scheduler-assisted prefetching mechanism. This mechanism is a general solution to the page fault overhead problem without limitations on hardware and task set. Our approach is to modify scheduler to predict task sequence and modify virtual memory manager to accumulate frequently used pages. The lists of pages are gathered for each task by monitoring page faults. For each task its frequently used pages are prefetched before it gets scheduled for execution.

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

There have been several other prefetching mechanisms. The naïve one-block look ahead prefetching [13] is certainly useful but does not work for tasks with non-linear data access patterns. Other works [10, 14, 15, 16, 17] try to implement prefetching when the virtual to physical addressing transformation takes place. They consider architecture-specific optimizations of cache use. For this reason the set of devices where such approaches are applicable is limited. The mechanism suggested by Lin, et al. [12] can be applied only for embedded systems running a limited predefined set of application with a given set of execution traces.

In 2001, Suh, et al. [1] and Chiou, et al. [2] presented results of simulation of an approach, which is similar to scheduler-assisted prefetching. Their work showed theoretical decrease in number of page faults. They theoretically evaluated if it would be practical to develop such prefetching mechanism. However, they did not develop a prefetching mechanism and did not implement it for any operating system.

It is also worth clarifying the difference between scheduler-assisted prefetching and two recently created prefetching mechanisms: the SuperFetch [19] implemented in Microsoft Widows Vista and the Swap Prefetch [18] implemented in -ck patches of Linux. The SuperFetch and Swap Prefetch are idle time prefetching techniques. Their main goal is to resolve so called “after lunch syndrome.” When one leaves working computer idle for 30 minutes during the 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. On the contrary the scheduler-assisted prefetching works when the system is under heavy memory load and there are several tasks 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 experiments set-ups, goals, benchmarking tools and the
results. Finally, Section 6 summarizes and concludes this paper.

### 2 Problem Definition

As it was mentioned earlier, page faults force task spend part of its CPU time waiting for a page to be loaded. Figure 1 illustrates page fault overhead.

![Page fault overhead](image)

**Figure 1: Page fault overhead.**

If the faulted page is a discarded code page, then it is loaded from the executable binary on a storage device. If it is a swapped out data page, then it is loaded from the swapping space. The code page faults became typical for embedded systems after they started using a cheaper NAND flash memory where XIP is impossible. The data page faults are also common since modern embedded systems have advanced multimedia functionality.

The majority of such page faults happens in a short period after rescheduling as task restores the state, in which its last job finished execution. Hence our general goal is to decrease the number of page faults by prefetching pages which are currently absent from memory and are likely to be accessed after the next context switch. Several problems should be solved in order to make prefetching efficient:

1. The task that will be executed next has to be determined.
2. The pages, that are likely to be accessed by the next task should be determined.
3. The pages in RAM, that can be substituted by prefetched pages without degradation of the performance of the current task have to be chosen.
4. The correct moment for starting the prefetching during current job execution has to be calculated.

In this section we formally define the four problems addressed by our scheduler-assisted prefetching. Let a system run $N$ tasks $\{\tau_1, \tau_2, \ldots, \tau_N\} = T$. Traditionally jobs are numbered with respect to a task. However for our problem the sequence of jobs formed by scheduler is more important. Therefore we will use index $j$ to number jobs generated by scheduler. This means that jobs $j, j+1$ are the instances of different tasks executed one after another.

#### 2.1 Task Prediction Problem

Let us consider scheduler as a function $S : 2^T \rightarrow T$, which determines the next for execution task among ready to be released according to some algorithm. Now the task prediction problem is to use or modify $S$ to get $\hat{S}$ so that prefetcher can get $(\tau_k, \tau_{j})$ which will be rescheduled one after another: $\hat{S} : 2^T \rightarrow T \times T$.

#### 2.2 Page Choosing Problem

Consider a task which has one of its job scheduled as $j^\text{th}$ job and set of pages $W_j$ which were swapped out or discarded by the release time of $j^\text{th}$ job. The page choosing problem for this task is to select subset of $W_j$ consisting of pages that will be accessed during $j^\text{th}$ job execution.

#### 2.3 Page Replacement Problem

Usually, an operating system has a default page replacement policy. However the scheduler-assisted prefetching change page access pattern. Therefore it is required to answer the following questions:

1. How does the typical page replacement policy behave when scheduler-assisted prefetching is enabled?
2. Can the scheduler-assisted prefetching cause regression if we use the default replacement policy?

Answering the first question, we analyse impact of scheduler-assisted prefetching on a 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 suggest changes to the policy, that allow to avoid such regression.

#### 2.4 Activation Moment Problem

The activation moment is a moment $t_{j^{\text{pref}}} \in (r_j, r_{j+1})$, where $r_j, r_{j+1}$ are consequent release times of some consequent jobs. By varying this parameter we vary the set of faulted pages of job $j+1$ and set of swapped out and discarded pages of job $j$. The activation moment problem is to choose such a moment $t_{j^{\text{pref}}}$ that minimises number of page faults of the next job and page displacements of the current job.

### 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 2 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, there are four questions to answer while designing the prefetching system. These questions are covered in the next subsections.

3.1 Task Sequence Prediction

In order to do the prefetching as it is shown on figure 3, we need to know the task which will be executed next after the current task.

![Figure 3: Next task prefetching.](image)

We modify the task scheduler, so that it calls original scheduler two times to provides us with a task which is planned for execution next as it shown by activities a, b and c. Planned means that this task will be rescheduled for execution unless some more important task appears in a run-queue. The modified schedule can be expressed mathematically as follows:

\[ \tilde{S}(X) = [S(T), S(T \setminus T)] \]

Usually, the 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 this 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 the list, present in memory as activities d, e and f show.

Of course at the rescheduling moment scheduler may return task which is different form the task selected for prefetching because a new task has been started or temporally suspended task was waken up. In that case prefetcher will not give any performance boost, but will not degrade it as well for the reasons discussed in Section 3.3.

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

3.2 Prefetching Pages Accumulation

Trying to prefetch all pages of a given task is impractical, because sum of the memory footprint of the current and next tasks 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 next job execution differ insignificantly [20]. It means that working set changes slowly and gradually as figure 4 shows.

![Figure 4: Page access pattern.](image)

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

### 3.3 Replacement Policy Analysis

In this subsection we assume that 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 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, new page is allocated and added to the head of this list. Since a plain implementation of this algorithm is impractical, modern operating systems typically use algorithms which mimic 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 pages of this tasks are actually least recently used. Figure 5 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 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 5: LRU under heavy memory load.](image)

The extreme case of this situation is when the system progresses very slowly because of 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 pages, which are likely to be accessed, be present in memory, these pages also move to the beginning of the LRU list. The more accurate approach is to artificially put prefetched pages in 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 will guarantee no performance degradation. Figure 6 illustrates 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 6: LRU behaviour with prefetching.](image)

### 3.4 Activation Moment Calculation

The last 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 handler for timer. We set up such a timer during rescheduling. This timer will interrupt the 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 to initiate prefetching during scheduler tick. Usually, during scheduler tick we can calculate how much time left before the rescheduling execution and based on this information 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 the left border of the task execution period – the moment, when the job is released, while the last approach puts that moment relative to the right border – the moment, when 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 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 in different schedulers and even in different versions of the same scheduler. The second method does not have this disadvantage, but assumes that scheduler uses scheduling ticks. In our implementation...
we used the tick-based approach.

Now we discuss how we determine the right moment for the prefetching activation. Let us introduce variable parameter $\xi \in [b, 1]$ which determines what fraction of current job should be completed before we start prefetching of the next job. The bound bottom value $b \in (0, 1)$ determines given system earliest possible prefetching. Smaller values of $\xi$ corresponds to earlier prefetching, while $\xi = 1$ means that no prefetching is required. We can calculate each moment of prefetcher activation using the following formula:

$$
T_{j+1}^{pref} = r_j + \xi(r_{j+1} - r_j).
$$

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 $\xi$ 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 this 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 list of recently faulted pages, we had to modify memory map structure and routine responsible for its creation. These changes allow us to accumulate 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);
}

The Linux kernel distinguishes several kinds of page faults due to several possible reasons they are initiated and ways they are processed. 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. We count page faults, because we will need this number during experiments.

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 reduce 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 5. 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 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 of RAM
- 256 MB of Swap
- Gentoo Linux Minimal

The 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. 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. Since Linux kernel 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 effect of prefetching when system has enough memory for a given task set. This gives us one more ground for overloaded system consideration.

For the interactivity test we used **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 simulations and loads.

5.3 Experimental Results

The results of the Stress test are showed by figures 7 and 8. The figure 7 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 the contexts switch happens. Obviously, starting prefetching just after the 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 8 shows this negative effect. Of course non-major page faults require much less handling time, but still they interrupt execution of the 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.

---

\(^1\) Vanilla kernel – the Linux source tree released by Linus Torvalds without any other modifications.
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 1: Change of PF rate after prefetching.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Mjr. PF Change (%)</th>
<th>Total PF Change (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No prefetching</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prefetch-75</td>
<td>-30</td>
<td>-25</td>
</tr>
<tr>
<td>Prefetch-50</td>
<td>-37</td>
<td>+40</td>
</tr>
</tbody>
</table>

During our tests, we did not try to find an optimal moment for prefetching because, this value heavily depends on 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 the problem of performance degradation because of high rate of page faults. 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 scheduler-assisted prefetching. The decision which pages to prefetch is made with 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 answers four main problems of prefetching.

The task prediction problem can be addressed by modifications of scheduler so that it gives two next tasks instead of one. As a solution of the page choosing problem we suggest to accumulate limited list of last faulted pages for each task. Analysis of the page replacement problem showed that LRU with few modifications can be applied. Finally, we address the activation moment problem by choosing a fraction of time unit after execution of which the prefetching should be activated.

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 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 implemented prefetching on top of the Completely Fair Scheduler of the Linux kernel and performed experiments. We conducted two kinds of experiments: Stress test of the virtual memory and Interactivity experiment. The Stress experiment showed 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%, 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 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 utilization.

References

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,

performance by the addition of a small fully-
associative cache and prefetch buffers,” In Proc.
17th Ann. Int’l Symp. on Computer Architecture,

directed prefetching in scalar processors,” 25th Ann.
Int’l Symp. on Microarchitecture, pages 102–110,

prefetching for secondary caches,” International
Conference on Parallel Processing, pages 314–323,
1997.

Int’l Conf. on Parallel Architectures and

on High-Performance Computer Architecture

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


[20] Abraham Silberschatz, Peter Galvin, Greg Gagne,
Applied Operating Systems Concept, John Wiley &