Anticipatory Paging Algorithm Based on Code Page Prediction

Jinfeng Liu  jinfengl@uci.edu  ID: 65547013

Abstract

The objective of this project is to suggest an anticipatory paging algorithm that can reduce the page fault rate at a tolerable overhead. Demand paging and LRU-approximate page replacement algorithms are commonly accepted by existing systems. These algorithms are based on the “past” knowledge by looking back to the run-time history. On the other hand, the optimum paging scheme should make decisions based on the program behavior in the future. Although such optimum algorithm is impossible to achieve, we can try some heuristics to predict the near future of program execution based on history. Therefore, we propose an anticipatory paging scheme to make prediction to the next code page of the current code page.

The access to data pages and the execution on code pages have different behaviors. In general, the program behavior to access data pages is hardly predictable. Prediction-based anticipatory paging may not yield any performance speedup. On the other hand, the behavior of accessing code pages is of more regularity. Giving the current code page that is being executed, the next code page to be executed could be one of several specific pages -- we introduce the concept next page set that contains the code page numbers of those. Thus, the goal of our anticipatory paging algorithm is to select the next code page of the current page from the next page set that is most likely to be “hit” by the actual next page.

The program behavior of SPEC95 benchmarks is studied. The algorithm to predict the next code page is implemented by program run-time history and heuristic methods. For the SPEC95 INT benchmarks, by using our predictor the page fault rate can be reduced by 10% - 20%. On the contrary, for the SPEC95 FP benchmarks, the performance speedup is slightly seen. This is because the floating-point benchmarks are intensive at data page accesses. Our prediction on code page access cannot significantly reduce the overall page faults.

Not much work has been done in this area by examining the up-to-date benchmarks. We hope our research can bring some insights into the anticipatory paging algorithms.

Keywords: anticipatory paging, prefetching, data page, code page, prediction, next page set, page fault rate

1. Introduction

Most contemporary computer systems employ virtual memory management by using either paging or segmentation methods, or their combinations. One underlying principal is that we use the virtual memory management to hide the limited physical memory space to application program. Thus, the programmer can use as large as the virtual memory space allows without knowing the actual size of physical memory. Different software and hardware mechanisms are developed to support an effective mechanism, such as TLB, page table, and etc. One of the major barriers to achieve high performance is the page fault. When application program requires more memory than how much it is allowed to have, page fault will take place and the number of page fault goes up abruptly at limited physical memory.

One scheme to reduce the page fault is to utilize an effective page replacement algorithm. LRU-approximate page replacement algorithms are widely accepted by existing systems. These algorithms are based on the “past” knowledge by looking back to the run-time history. Page replacement algorithms have been studied for decades. In most cases the replacement algorithms are combined with demand paging mechanism that only brings a new page into main memory at a page fault. However, to achieve an optimum performance, the paging mechanism should make decisions based on the future program execution. If a new page can be brought into main memory before the page fault occurs, the page fault rate can be minimized. Such optimum algorithm is difficult to
achieve because the future program behavior is hardly predictable. Plus, such mechanism also increases both hardware complexity and software overhead, but the performance speedup is not guaranteed. The anticipatory paging mechanism is not commonly utilized, and it is not widely discussed in research area. Up to now, a few researchers are still exploring the better performance of LRU-based replacement algorithms [5][6], while the sources to anticipatory paging mechanism is very limited.

However, in this project, we try to use some simple heuristics to predict the near future of program execution. We propose an anticipatory paging scheme based on prediction to next code page of the current code page; then prefetch this predicted page before page fault can occur. We examine the SPEC95 benchmarks, which represent the behaviors of most contemporary computer programs. The performance of our algorithms is also measured by simulating the execution of SPEC95 benchmarks. We have not seen much previous work that has been focused on this area by using the most up-to-date benchmark suits. We hope our study can produce some insightful ideas to the future paging systems.

Before the performance of paging mechanisms is measured, certain behaviors of program should be recognized. As it is commonly known, the accesses to data pages and the executions on instructions in code page have different behaviors. In general, the program behavior to access data pages is largely dependent on the programmer. Thus, the accesses to data pages are hardly predictable. Prediction-based anticipatory paging may not yield any performance speedup for data pages.

On the other hand, the behavior of accessing code pages is of more regularity. This is due to the locality brought by the sequential nature of the program. If all code within the current page consists of only sequential instructions, the next page is always the following logical code page. However, there are different types of non-sequential instructions, e.g., branch, jump, procedure call and return. In these cases, the next required page could be one of several specific pages -- we introduce the concept next page set that contains the code page numbers of those pages that could be the next code page to execute. Thus, the goal of anticipatory paging is to select the next code page of the current page from the next page set that is most likely to be “hit” by the actual next page.

We assume part of the program will be executed multiple times. Initially, the next page set of all pages only contains the following logical code page. This assumes a sequential nature of program execution. This field is changed dynamically by the run-time history. When the same page is executed again, such knowledge will be helpful to predict the future. Each number in the next page set should be associated with a priority. The page with highest priority will be selected as the candidate for next page to bring in. The multiple page numbers and the corresponding priorities can be changed by different criteria adaptively during run-time execution. In order to make our predictor easy to implement, we discuss the schemes that repeat the next code page in last execution or select the most frequent next code page.

An important issue is related to pages containing procedures. Since a procedure should return to its caller, the page containing its caller instruction should be assigned a high priority in the next page set. But this will incur problems. A procedure can be called multiple times in anywhere of the program, this will cause the next page set to overflow. A heuristic is to remove the caller’s page number from the set after the procedure returns. Another problem is that we only want to have the caller’s page number in the next page set of pages that contain the return instruction. In the ensuing part we discuss how to maintain such information and make prediction effectively.

The performance of the algorithms is simulated on SPEC95 benchmarks. Improvement at 10%-20% in reducing the page faults can be achieved. Our algorithm only works on code pages. Therefore, for the SPEC95 FP benchmarks that have intensive data accesses, the prediction may not yield much performance speedup. This can be seen in our simulation.

If there is no free page available, page replacement takes place. Our next page set model can affect the existing LRU or its approximations. A general heuristic is not to swap out any pages that are among the next page set of the current page. In this approach, even if our prediction is wrong, the actual next page still has some chances to “hit” the existing pages in main memory. Some related techniques are discussed.
One of the difficulties to make good prediction is one page can have many outcomes as its next code page. This is decided by the non-sequential instructions in the page. A general heuristics is small page can limit the number of outcomes, thus produces more predictability. We also discuss this issue.

The rest part of this paper is organized as follows. Section 2 introduces the algorithm. Section 3 gives the results on the simulation and explains how our algorithm can improve the performance. Part 4 discusses some other related issues. And in Part 5 we make a conclusion and suggest directions for further study.

2. The algorithm to predict next page

The goal of our study is to reduce the page fault rate. If the physical memory space is larger than the amount that a program requires, the number page fault equals to the total required pages. In this case little speedup is possible. In our study, all programs are allocated less physical memory pages than they require. To simplify the run-time execution complexity, we only execute one process at a time. That is, we only focus on how our algorithm affects the page fault rate of single process.

Therefore, during a program execution, some pages reside in main memory and some stay in secondary storage. We organize all pages resident in main memory in an LRU queue. The LRU queue is implemented in a form of double-linked list. In this way we can ignore the LRU reference field in a page table. Although LRU is rarely implemented in real system, it is still a proper example on the algorithm study.

A multi-level inverted page table is implemented. The table is indexed by the program counter (PC) by a direct-mapped method. To minimize the size of the table, the high-order bits of the PC are used as a tag field in the page table. In a conflict on the same page table entry, the new page replaces the old one and the old one goes to the second level. Each entry contains a tag, a pointer to the second level page table, a pointer to the memory pages, a flag field indicating the attribute of a page (data, code, dirty, valid, and etc.), pointers in LRU queue, and a next page set for code pages. Again, to concentrate on paging algorithm only, we ignore factors incurred by TLB, TLB misses, and page table misses.

The major complexity of LRU is that the LRU priorities are changes at each memory access. In our model, the priorities are represented as the LRU link pointers. The easiest way is to move (or add, on a page miss) every newly accessed page to the LRU head. This is less efficient since during a certain period, only one code page contains the currently executed instruction. In case of frequent memory data access, the LRU head is changed frequently from the accessed data pages to this single code page. In order to minimize the operation on LRU queue, we make this page always on the LRU head. The currently referenced data page will be moved to the second node of the list.

In order to predict the succeeding code page, we maintain a next page set in each entry of page table for a code page. The set contains an array of page numbers associated with some counters and pointers to keep the run-time history. During the program startup, there is no history information. When a code page is first brought into main memory, we predict the next logical code page will be executed. After a code page is executed once, we keep the history in its next page set and used the history to make prediction if this page is executed again.

Fig. 1 (a) On data page hit, data page will be moved to the 2nd node
Fig. 1 (b) On data page miss, data page will be inserted to the 2nd node.

Fig. 1 (c) On code page hit, new code page will be the LRU head, and the predicted next page will be inserted at the 2nd node (whether this page is in main memory or not).

Fig. 1 (d) On code page miss, new code page will be added to the LRU head, and the predicted next page will be inserted at the 2nd node (whether this page is in main memory or not).
The prediction to next code page is made at each time a new code page is executed. The new code page will appear at LRU head. The predicted page will be placed at the second node of LRU. Fig. 1 (a)-(d) depicts the page placement scheme in terms of data page hit (a), data page miss (b), code page hit (c) and code page miss (d).

If the LRU queue is full, that is, the maximum page number of this process is reached, the node on LRU tail will be the victim to be swapped out. The replacement can take place in (1) a data page miss, (2) a code page miss, and (3) the predicted code page is not in main memory.

On a new code page access, we put the new page on the LRU head, and we always put the predicted next code page to the 2nd node. This gives the predicted page maximum opportunity to be resident in the memory till it is executed, thus the maximum possibility to hit this page is secured, if the prediction is correct. On the other hand, if the prediction is wrong but this predicted page would also be executed in near future, it still has a better chance to be hit on its next execution. An extreme case on misprediction is that this predicted page would not be executed for long. In such case, a non-useful page stays in the beginning of the LRU queue; in another word, it has a higher priority. Therefore we lose one useful page in main memory in a relatively long period until this mispredicted page is replaced, the penalty can be high.

There can be different schemes to predict the next code page of the current code page. The simplest way is to repeat the last execution to its code page. This does not require any hardware support and software overhead in only one extra field for a page number in the page table. The performance is normally not guaranteed. Another approach is to select one page of its next page set that is executed most frequently. The next page set contain a certain number of pages that have been executed in the past. The prediction will choose the page that is executed most times. When the next page set is full, the new page will replace the one that is least frequently accessed. This also needs no additional hardware but requires more space in each page table. As we can imagine, these two strategies may not yield excellent prediction accuracy but the overhead is slight only on software.

A sophisticated scheme is to employ a history buffer in page table to keep the latest execution history. A decision-making logic (such as a finite-state-machine) can be associated with the history buffer to determine the prediction. For example, if the instruction flow transitions from a certain code page to other code pages A, B, and C in a repetitive pattern of AABCAABC……, the last-time predictor will achieve 25% accuracy, the most-frequent predictor will have 50% accuracy, while if we have a more sophisticated predictor with a 4-entry history buffer, the prediction accuracy can be close to 100%. The buffer should be long enough so that the history can be more complete. If the pattern changes to ABCBAC……, the 4-entry history buffer will not work. Generally, this approach can achieve high accuracy but the overhead is also high.

We should pay special attention to the program behavior in terms of procedures. A procedure should return to its caller site in most cases. When a procedure is about to return, normally the caller site is in a different page and that page is highly predictable, since the return address is in the system stack. Therefore, for code pages that contain procedures, we should use different strategy to predict its next code page. We can certainly keep the page of the caller’s site in memory. However, if the procedure is large, keeping the caller’s page for a long time before it returns can reduce page fault by one when it returns, but it may even increase the page faults during this long period. We want to make this prediction not too long before the procedure is going to return. Therefore, the PC of the return instruction must be kept to make this decision. We implement a procedure table to store information on all procedures. Each entry in this table is related to a procedure in the program. The entry contains procedure entry and exit addresses. The procedure table is built up during run-time execution. In order to look up the procedure table, the index to the table also needs to be pushed into the stack. We implement this modified version of system stack in return address stack. Fig. 2 shows the algorithm to predict the caller’s page of a procedure.

In Fig. 2, page A calls procedure printf() starting in page X. During program execution in page X, since PC and procedure return instruction are not in the same page, next code page prediction is made by normal schemes. When instruction reaches page Z, at this point PC matches the return instruction in the same page, then the prediction is made to page A, which is the caller of procedure printf().
This strategy on procedure return address prediction can improve the prediction on procedure execution. However, it will suffer from the procedures that have many different return sites. For example,

```c
int printf()
{
    for(i = 0; i < 100; i++)
        if (condition)
            return 0;
    return 1;
}
```

If the first return instruction within the loop stays in page Y, it would be difficult to make an accurate prediction, since such information is hard to maintain in the procedure table and run-time decision is not easy to make.

![Diagram of procedure return address prediction](image)

We implemented the simplified prediction schemes on last-page prediction and most-frequent page prediction. We also combine both schemes with the procedure return address prediction. The simulation results are presented in next part.

3. Simulation results

The simulation environment is based on SimpleScalar tool set version 2.0. Memory page size is 4KB. Inverted page table has 32K entries in first level. Instruction length is 8 bytes. There is no paging mechanisms in the SimpleScalar toolset. Our simulation is made on paging algorithms explained in section 2.

We use SPEC95 benchmarks to evaluate our paging algorithm. SPEC95 benchmarks are a series of programs that exhibit various behaviors. We chose six integer benchmarks from SPEC95 INT suite and five floating-point benchmarks from SPEC95 FP suites. Some attributes of these benchmark programs are listed in Table 1.

Different paging schemes are simulated for each benchmark. To compare our predicted anticipatory paging algorithm, we also simulate demand-paging scheme with LRU replacement algorithm. In order to explore how
much improvement we can have by using prediction, we assume one scheme that can achieve perfect prediction. The number of pages allocated to each benchmark varies from the total pages it needs to as few as 10 or 20 pages. Fig. 3 shows the number of page fault of gcc by using demand paging, last-page prediction, most-frequent page prediction and perfect prediction.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>No. of Instruction</th>
<th>No. of data pages</th>
<th>No. of code pages</th>
<th>No. of data page accesses</th>
<th>No. of code page accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>1.26G</td>
<td>562</td>
<td>509</td>
<td>512M</td>
<td>36.9M</td>
</tr>
<tr>
<td>compress</td>
<td>3.64M</td>
<td>141</td>
<td>23</td>
<td>2.2M</td>
<td>47.8K</td>
</tr>
<tr>
<td>go</td>
<td>548M</td>
<td>131</td>
<td>145</td>
<td>157M</td>
<td>16.0M</td>
</tr>
<tr>
<td>jpeg</td>
<td>555M</td>
<td>1135</td>
<td>76</td>
<td>142M</td>
<td>4.87M</td>
</tr>
<tr>
<td>lisp</td>
<td>11.5M</td>
<td>27</td>
<td>43</td>
<td>5.47M</td>
<td>575K</td>
</tr>
<tr>
<td>perl</td>
<td>10.5M</td>
<td>33</td>
<td>112</td>
<td>4.70M</td>
<td>495K</td>
</tr>
<tr>
<td>fpppp</td>
<td>1.87G</td>
<td>45</td>
<td>72</td>
<td>994M</td>
<td>85.5M</td>
</tr>
<tr>
<td>hytrod2d</td>
<td>975M</td>
<td>2170</td>
<td>52</td>
<td>257M</td>
<td>12.6M</td>
</tr>
<tr>
<td>su2cor</td>
<td>1.05G</td>
<td>2038</td>
<td>64</td>
<td>347M</td>
<td>15.6M</td>
</tr>
<tr>
<td>swim</td>
<td>850M</td>
<td>3612</td>
<td>42</td>
<td>264M</td>
<td>27.1M</td>
</tr>
<tr>
<td>tomcatv</td>
<td>2.80G</td>
<td>3612</td>
<td>41</td>
<td>869M</td>
<td>89.1M</td>
</tr>
</tbody>
</table>

Table 1. SPEC95 benchmarks used in simulation

As it can be seen from Fig. 3, the number of page fault increases sharply when the number of available pages drops. Using perfect prediction, the number of page fault can be reduced considerably. While last-page prediction and most-frequent page prediction do not yield a marked difference. Since the number of page faults varies in a large range, it is difficult to show the comparisons for all schemes in this manner. For the rest chart, we use the page fault rate in percentage of the number of page fault on demand paging. We make demand paging as our base model, and measure how much improvement we can achieve by reducing the number of page fault in percentage.

Fig. 4 (a) – (f) illustrate the page fault rate of integer benchmark programs. Fig 5 (a) – (e) show the page fault rate of floating point benchmarks. Page fault rate is relative to number of page fault in demand paging scheme in percentage. The line 100 represents how many page faults occur in demand paging. The prediction accuracy is marked with the legends corresponding to each prediction method.

In the simulation results to integer benchmarks, perfect prediction can yield about 40%-60% dropping of page faults. While using last-page or most-frequent page prediction, the improvement is around 10% - 20%. This is not surprising since the achieved prediction accuracy is around 50% - 60%. One exception is jpeg, although the
prediction accuracy is high, the improvement is not seen when the available pages exceed 200. Even the perfect prediction does not yield any exciting results. Table 1 tells the answers to such behavior. In ijpeg, data access is much more intensive. The number of page fault on data pages should be far more than that can happen on code pages. In this case, our prediction on code page execution contributes little on the overall improvement. Normally when the page number is big enough, the number of page fault remain quite constant regardless difference between paging schemes. This behavior is proved by our simulation on ijpeg and more float-point benchmarks. For simplicity the data on number of page fault is not shown. We continue focus on the relative page fault rate.

Fig. 4 (a)  
Benchmark: gcc  

Fig. 4 (b)  
Benchmark: compress  

Fig. 4 (c)  
Benchmark: go  

Fig. 4 (d)  
Benchmark: ijpeg  

Fig. 4 (e)  
Benchmark: Lisp  

Fig. 4 (f)  
Benchmark: perl
The behavior of float-point benchmarks is completely
different compared to integer benchmarks. It seems not
possible to achieve any performance speedup since
even the perfect prediction can only reduce page fault
up to 0.2%. In Table 1, we find that the floating-point
benchmarks are mostly intensive on data accesses.
Among all the required pages, only around 50 are code
pages, the rest 2000 pages are all data pages. This
makes our prediction on code pages too trivial to affect
the overall page faults, since the number of page fault
on data pages is huge. One exception is fpppp, to our
surprise, an improvement of 40% is accomplished. The
reason is that this benchmark is the only one that does
not have intensive data accesses. In fpppp, the numbers
of code pages are more than data pages. This accounts
for the fact that a low-accuracy prediction can yield
considerable improvement.
From the simulation results so far we can see the potential speedup by prefetching the predicted code page in integer benchmarks. For floating-point benchmarks, since they are mostly data access intensive, the prediction on code page does not affect the page fault numbers effectively.

To further explore the opportunity of how much the prediction can reduce the page faults, we employ procedure return prediction discussed in previous section. The simulation results are shown in followings. Fig. 6 (a) – (f) describe the simulation result of integer benchmarks. Fig. 7 (a) – (e) depict the results in floating-point benchmarks.

In integer benchmarks, the procedure return prediction produces extra 1%-5% accuracy in prediction. Accordingly, the decrease in page fault rate is about 3%-10%. The procedure return prediction does not significantly boost the prediction, since the procedure call and return instructions cannot be dominant throughout all instructions in programs. Moreover, the predictions made to procedure returns are based on heuristics. Program statistics show that the accuracy for the procedure return prediction is around 70%-90%. Our decision to bring the page containing caller site is made when PC is in the same pages as the exit point of a procedure. This is not accurate for all conditions. However, by using more confined heuristics, e.g., bring in the caller’s page when PC is close enough to procedure exit point, we can have better
accuracy but also incur more limitations. That is, fewer pages can be applied to such strict heuristic conditions. In order to keep our algorithms suit for general cases, we still use the loosely confined conditions for heuristics.

**Fig. 6 (e)**

Benchmark: lisp

- Most-frequent page prediction 58.1%
- Most-frequent page & procedure return prediction 58.7%
- Last-page prediction 53.3%
- Last-page & procedure return prediction 54.0%

**Fig. 6 (f)**

Benchmark: perl

- Most-frequent page prediction 65.1%
- Most-frequent page & procedure return prediction 67.2%
- Last-page prediction 52.3%
- Last-page & procedure return prediction 56.6%

**Fig. 7 (a)**

Benchmark: fpppp

- Most-frequent page prediction 57.9%
- Most-frequent page & procedure return prediction 68.3%
- Last-page prediction 69.6%
- Last-page & procedure return prediction 61.6%

**Fig. 7 (b)**

Benchmark: hydro2d

- Most-frequent page prediction 76.7%
- Most-frequent page & procedure return prediction 69.5%
- Last-page prediction 72.3%
- Last-page & procedure return prediction 66.7%

**Fig. 7 (c)**

Benchmark: su2cor

- Most-frequent page prediction 65.2%
- Most-frequent page & procedure return prediction 61.5%
- Last-page prediction 65.0%
- Last-page & procedure return prediction 62.7%

**Fig. 7 (d)**

Benchmark: swim

- Most-frequent page prediction 80.1%
- Most-frequent page prediction 64.8%
- Last-page prediction 72.8%
- Last-page & procedure return prediction 61.4%
There is very little improvement in floating-point benchmarks, as it is expected. For such data access intensive programs, the prediction to code page execution cannot affect the number of page faults effectively.

So far, our study on anticipatory paging will work well on programs where the data access is not intensive. For the SPEC95 benchmarks, when the number of code pages and the number of data pages are in balance, the simple prediction scheme can yield 10%-20% improvement in reducing the page fault.

In next section, we discuss other factors that can affect the number of page fault.

4. Other issues

4.1 Page replacement strategy

In all our simulation above, LRU is used for replacement an old page with the new one. There can be other strategies. One natural idea is to select a least-recently-used clean page from the LRU queue. The idea is to minimize the write-back to hard disk. By simulating this scheme, the result shows the sharp increase in page faults, as shown in Fig. 8. This is because the clean pages in memory are likely to be swapped out, finally there are very few code pages in memory. Thus, the page miss on code pages grows up abruptly. Therefore, the policy to victim the clean page is not desirable.

Based on our next page set model, another heuristic is not to swap out a page that is in the next page set of the current code page. This can be expensive since at each replacement, the LRU queue is searched from the tail to find the first mismatched page to the pages in the next page set. We also simulate this strategy. Fig. 8 shows the result of running gcc by using last-page prediction. A comparison is made to last-page prediction with LRU replacement. The new method is only better at an improvement of less than 0.5%. Therefore, this strategy is not desirable due to the high overhead.
4.2 Data page prediction

After we try to predict next code page, a natural thought is to predict a data page of the next code page. The predicted data page could be the most-frequently accessed data page of the predicted code page. If we can achieve high prediction accuracy on the next code page, we may expect its most-frequent data page should be always accessed. The simulation is made to evaluate this approach. Similar to the data structure of next page set, we implement a data page set of each code page. When a code page is chosen to be the predicted next code page, we also bring its most-frequent data page into main memory. The simulation result of gcc is illustrated in Fig. 10.

As shown in the results, there is very little speedup even if we make the perfect prediction. When the prediction is not 100% accurate, the performance becomes worse. This is somehow against our intuition, since the perfect code page prediction plus most-frequent data prediction should yield an even better result.

The statistics on benchmark gcc show that the average instructions being executed during one code page execution are only 34, that is, in average a code page can only execute 34 instructions before the execution transitions to another code page, while the whole code page in size of 4K bytes contains 512 instructions. This is due to the high frequency of branch, jump and procedure call and return instructions. Therefore, the most-frequently accessed data page may vary from time to time during different dynamic executions of a code page. Even when we know for sure one code page is going to be executed, the data it is going to accesses is still hardly predictable. In gcc, the total data pages that one code page has accessed are over 50. Among these pages, the most frequently accessed pages only account for 17.5% in percentage. This would be the reason why data page prediction can hardly work.

4.3 Page size

We are also interested in how the size of memory pages can affect our predicted paging algorithm. We want the number of candidates in next page set to be as few as possible, thus high-accuracy prediction can be achieved. Assuming page 1, 2, 3, 4 are logically continuous pages. Consider Page 2, there are three possible candidates for its next page, 1, 3 and 4. If we use large page size, Page 1 and Page 2 are combined as a single page as Page A, so are Page 3 and 4 as Page B. Then the next page of Page A is always B, which is 100% predictable. This implies coarse granularity on page size. However, we can also make the page smaller. If we maintain the page size in a fine-grained manner so that each page only contains one branch instruction, e.g., page a, b, and c as shown in the figure, the next page should be either the page that contains the branch target or the following page. In this approach, we can achieve high prediction accuracy as well. According to our intuition, small pages may be better, since larger pages will also contain more non-sequential instruction. Plus, for the code pages that have many procedure calls, smaller page size may be desirable, regardless of more overheads.

The simulation we have made so far is based on the page size of 4K bytes. In order to measure how the page size can affect our paging algorithms, more simulations are made on page size of 2K bytes, 8K bytes and 16K bytes. For simplicity, we chose 3 benchmarks to compare the results. Fig. 12 (a) –(d) demonstrate the simulation results of go at three prediction schemes, perfect prediction, last-page prediction and last-page prediction with...
procedure return prediction, when page size varies from 2K to 16K. Fig. 13 and 14 show the result of perl and su2cor in same conditions respectively.

In Fig. 13, 14, and 15, it can be seen that similar curves repeat in page size of 2K, 4K, 8K and 16K. It proves that our algorithms are consistent with different page sizes. In go and perl, when page size increases, the prediction accuracy also rises. In su2cor, the prediction accuracy also increases in large page size. An exception is that the prediction on procedure return causes the prediction accuracy to drop. This behavior is repeated in page size of 2K, 4K and 8K. When page size is 16K, the accuracy is same at 78.5%. The large page size may hide the discrepancies of two prediction schemes.

An important observation is that although the prediction accuracy goes higher with larger pages, the performance, on the contrary, becomes a little worse in larger pages. (This can be told by the data, but less obvious on the chart.) The question is why we achieve lower performance in higher prediction accuracy. One reason would be brought by the assumption of our algorithms. Our algorithms bring the predicted page at the earliest time – right after the new code page is first accessed. In the larger pages, this predicted page would wait for a longer time and remain not hit till it is executed. During this period, one page is actually “lost”. Thus, the rate of page fault rate goes high.
Fig. 12 (a)  
Benchmark: go  
Page size = 2K

No. of pages  
Page fault rate  
Perfect prediction 100  
Last-page prediction 46.3%  
Last-page & procedure return prediction 53.4%

Fig. 12 (b)  
Benchmark: go  
Page size: 4K

No. of pages  
Page fault rate  
Perfect prediction 100%  
Last-page prediction 57.3%  
Last-page & procedure return prediction 61.6%

Fig. 12 (c)  
Benchmark: go  
Page size: 8K

No. of pages  
Page fault rate  
Perfect prediction 100%  
Last-page prediction 58.9%  
Last-page & procedure return prediction 63.0%

Fig. 12 (d)  
Benchmark: go  
Page size: 16K

No. of pages  
Page fault rate  
Perfect prediction 100%  
Last-page prediction 61.1%  
Last-page & procedure return prediction 65.4%

Fig. 13 (a)  
Benchmark: perl  
Page size: 2K

No. of pages  
Page fault rate  
Perfect prediction 100%  
Last-page prediction 51.2%  
Last-page & procedure return prediction 54.7%

Fig. 13 (b)  
Benchmark: perl  
Page size: 4K

No. of pages  
Page fault rate  
Perfect prediction 100%  
Last-page prediction 52.3%  
Last-page & procedure return prediction 56.6%
5. Conclusions and Future Work

Most contemporary computer systems employ demand-paging schemes in virtual memory management. It is commonly known that demand paging mounts little effort to minimize the rate of page fault. The reason for the prevalence of such design is that demand paging is easy to implement; and, on the other hand, the anticipatory paging mechanism is both expensive on implementation and less effective in predicting the future of program execution.

The sources of anticipatory paging mechanisms are very limited in both industry and academic world. Our work is to explore how much possibility that an anticipatory paging scheme can improve the performance of paging systems in the context of modern computer programs. Not much research work on this topic has been sufficiently discussed. We hope our study can yield some insights to the future paging systems.

In this project, we focus on an algorithm to predict code pages in program execution at run-time. We assume a prefetching mechanism that can prefetch pages into main memory without slow down the program execution. We also ignore other facts such as TLB and page table miss that can affect paging performance. Our study suggests the run-time behavior of code pages is highly predictable. Therefore, an effective anticipatory paging scheme will have a significant potential to improve the performance of a paging system.

The algorithms are made upon simple strategies to predict the next code page to be executed. We use the data structure next page set to contain the possible code pages to be executed next to the current code page. Prediction to the next code page is made by choosing a candidate from the set. We studied the prediction schemes using last-code page prediction and most-frequent code page prediction, combined with heuristics to predict procedure returns. The algorithms turn out a drop to the overall page faults at around 10%-20% in most SPEC95 integer benchmarks with minimum software overhead. These overheads include more fields in page table to contain page numbers of the possible next code page, a procedure table to contain procedure information at run time, and additional field in system stack to index the procedure table. Except for a prefetching device, additional hardware overhead is not necessary.

Our study also indicate that the maximum performance improvement with perfect prediction can reduce the page fault rate to a degree as low as 20%-40% in comparison to demand paging schemes. By utilizing more sophisticated predictor, the accurate prediction is possible. The predictor can be implemented by history buffer in page table with a decision making finite-state machine, or, be associated with the hardware branch predictor.

The performance of the algorithms is quite consistent in various page sizes. As we expected, the performance speedup is better for small sizes regardless of the high overhead to maintain small pages. We also discussed some other techniques that can affect the algorithm. Although most of them are not more effective than LRU, the study brings insights to the mechanisms of the paging systems.

The algorithms can be effective only on programs that are not data-access intensive. In most SPEC95 floating-point benchmarks, the prediction on code page helps very little to reduce the overall page faults because page faults on data pages are dominant. Our work also reveals that data page prediction is hardly practical even in situation of perfect code page prediction. This limited the validity of our proposed algorithm.

The future research on this topic can be addressed into two directions. First, an accurate predictor can be achieved by more in-depth study on run-time behavior of the program. By collecting the patterns of code page access history, the high accuracy prediction to code page is promising. On the other hand, our study can be furthered by examining the run-time overhead of the algorithm. The run-time overhead can be measured by number of pages that are swapped into main memory, and number of dirty pages that are written to secondary storage. Our current study shows that the perfect prediction has no additional overhead compared with demand paging, while the other schemes will incur more pages swapped in and out at a 10%-20% increase. In an
anticipatory paging system, the swap-in and swap-out of pages can be more frequent than the demand paging system. However, the prefetching should not make a considerable slow-down to the system. How much the system can tolerate from additional in-and-out pages is important to determine the value of an algorithm.

References: