1 Introduction

Increasing memory requirements for applications, e.g., machine learning, coupled with the slowdown of DRAM scaling , makes DRAM one of the costliest components in data centers, constituting as much as 30% of the entire cost . Thus prefetching is essential to reduce the number of page faults and improve application performance.

NEED TO ADD MORE CONTEXT HERE to explain why we need prefetching for page faults.

Prefetching is a technique used to reduce the number of page faults by predicting the pages that will be accessed in the future and fetching them into the cache before they are accessed.

Prefetching is a very rich field with many different approaches. Traditional prefetching algorithms use heuristics to predict the next page to prefetch. These heuristics are based on the recent access patterns of the application and guess the next page to prefetch based on these patterns using a pre-defined algorithm .

With the rececnt bloom of machine learning (ML) techniques, researchers have started to apply ML to the problem of prefetching.

WHAT ARE WE ACTUALLY PROPOSING?

2 Background and Related work

2.1 Heuristic Page prefetching

Page prefetchers are used in most modern operating systems to reduce the latency of page access from swap. Traditional algorithms prefetch based on sequential access to virtual addresses and are successful at fetching spatially related pages. One of the recent state of art prefetcher, Leap , improved traditional prefetching using majority trend detection to identify strided patterns; this makes Leap resilient to short-term irregularities in the memory access stream. Leap improved performance for many applications but cannot prefetch irregular accesses. Multiple researchs have tried to improve Leap's performance by addressing its shortcomings in

- prefetching irregular patterns,
- isolating the swap subsystem and memory access histories of threads, and
- coordinating memory accesses from the garbage collector and the application.

All the above resulted in the non-perfect prefetching performance of the prefetcher in our initial experiments.

HOW TO CONTRAST OUR WORK WITH TRADITIONAL PREFETCHING?

Studies has shown the need for more aggressive prefetching to reduce the number of page faults while scrificing some protability and memory bandwidth. Thus, we propose a machine learning based prefetcher that can improve the prefetching accuracy and reduce the number of page faults.

2.2 Machine Learning Prefetching

Past success of machine learning in cache eviction and cache prefetching has shown that machine learning can be used to predict page accesses patterns with high accuracy.

NEED MORE PREVIOUS WORKS

2.2.1 LSTM based prefetching.

LSTM, first introduced in 1997 , is a type of recurrent neural network that is capable of learning long-term dependencies.

3 Motivating study

To investigate which applications encounter pointer-based page faults, we developed a tool that analyzes a dynamic trace of register loads and page faults. We classify page faults into pointer-based and non-pointer-based and evaluate the performance of the default kernel prefetcher . Prior work performed a similar analysis to determine whether pointer chasing causes cache misses . However, the existence of pointer-chasing based cache misses does not guarantee the presence of pointer-based page faults. For example, cache misses within a page do not cause page faults; only cache misses across pages might cause them. Furthermore, page faults occur only when the application accesses swapped-out data.

3.1 Analyzer design

Our analyzer merges two traces: a dynamic trace of all values loaded into registers and a trace of page fault addresses. A pointer is simply a memory location containing an address; without type information, it is impossible to distinguish a pointer from a non-pointer. So, we borrow from prior work on cache prefetching to analyze dynamic application execution trace. If a value is loaded into a register from memory and is then used as an address or in an address computation, we consider it to be a pointer-based access. For example, consider the linked list traversal and its corresponding assembly shown in **??**. The

Workload	Source	WSS
Array streaming	Microbenchmark	50%
Random linked list traversal (LL)	Microbenchmark	50%
Canneal	Parsec	50%
xHPCG	HPCG	50%
BFS on twitter dataset	GapBS	25%
Redis benchmark LRANGE	Redis	25%

Table 1. Workloads used for analysis. They represent a diverse spectrum of pointer-based fault intensity as shown in **??**. WSS is the working set size in RAM

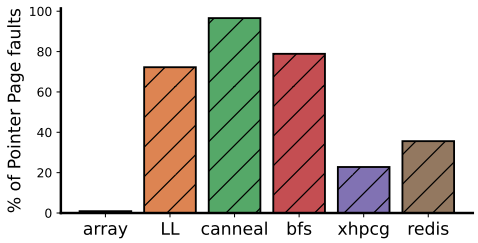

Figure 1. Percentage of pointer-based page faults for different workloads

value of *curr* stored on the stack, is first loaded into register `rax` (Line 2) and subsequently accessed to load *curr->next* from memory (Line 3). We classify this as a pointer-based access because the value of *curr* was initially loaded into a register and subsequently used as an address. If the access to *curr* caused a page fault, we classify that page fault as a pointer-based page fault.

```
1 ;curr = curr->next;
2  mov -0x600028(%rbp),%rax;load address of curr to rax
3  mov 0xff0(%rax),%rax;load the value of the curr->next
4  mov %rax,-0x600028(%rbp);store the loaded value
```

Listing 1. x86 Assembly for traversal

3.2 Analyzer implementation

We collect the dynamic trace of all values loaded into registers using Intel PIN (v3.26) and the page fault trace using `perf` on Linux (v5.19). We run the analysis on Linux due to the availability of Intel PIN and `perf`, but we expect the analysis approach to generalize across platforms. We gather these traces during one program execution. We merge the two traces using timestamps and process them in sequential order. The analyzer maintains two data structures; the current state of the execution in a registers-to-value map and a set of loaded values. Once all the registers containing a particular value are overwritten, we remove the original value from the loaded values set. When a page fault occurs, the analyzer checks if the page address was loaded into a register by checking the currently loaded values set and then classifies the page fault as pointer-based if it is present. The analyzer can analyze traces in both offline and streaming mode.

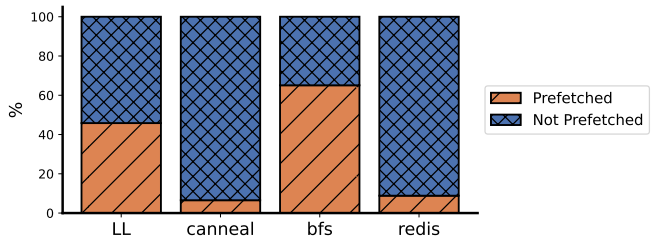

Figure 2. Accuracy of the kernel prefetcher on pointer-based page faults.

3.3 Pointer-based page faults

We analyzed different benchmarks (**??**). In order to evaluate the potential of CHERI-picking in ideally suited workloads, we looked for benchmarks that demonstrate pointer-chasing behavior. We chose benchmarks based on prior work having identified that they would benefit from pointer prefetching: Parsec's canneal , xHPCG and Redis . To have confidence in the validity of our tool, we also designed a linked list traversal microbenchmark, which dynamically allocated page-sized nodes and ordered them randomly to render the kernel prefetcher ineffective. For the array traversal microbenchmark, we allocated an array and streamed it sequentially twice. To focus on page fault behavior, we constrained the program's memory using `cgroups` .

?? presents our results. As we expect, a majority of the faults in the linked list microbenchmark are pointer-based, while none of the faults in the array streaming benchmark are. Unsurprisingly, pointer-based page fault rates vary in other workloads, ranging from about 30% for xHPCG to almost 100% for canneal. Canneal accesses elements by indexing into two lists of pointers, so we expect it to have a majority of pointer-based faults. The xHPCG benchmark maintains sparse vector objects and uses them to perform computation, leading to a lower percentage of pointer-based faults. The BFS benchmark performs breadth first search on a graph, stored in compressed sparse row format. Every access from a vertex to its children dereferences a pointer.

Our results suggest that some applications can benefit significantly from pointer prefetchers. An ideal pointer prefetcher should identify applications that can benefit from pointer prefetching without imposing overhead on applications that cannot, and achieve high prediction accuracy.

3.4 Performance of default kernel prefetcher on pointer-based page faults

The default kernel prefetchers detect sequential and strided accesses, make decisions quickly, and add little latency to the page fault path. Therefore pointer prefetchers should focus only on applications where the kernel prefetcher is ineffective. Figure **??** illustrates the accuracy of the default kernel prefetcher on page faults that were classified as pointer-based. As expected, the results show that kernel prefetcher performance also varies, thus pointer prefetchers should be used to predict only those page faults that the default prefetcher misses. In the case of the linked list traversal, the kernel prefetcher predicts approximately 50% of the pointer faults because the benchmark contains two loops; the first loop accesses the linked list nodes in the allocation order, while the second loop accesses them in random order. Accessing items in allocation order tends to produce a sequential access pattern that the default prefetcher handles well; accessing items in a random order produces an arbitrary pattern, so the default prefetcher fails. Surprisingly, although 78% of page faults for BFS are classified as pointer-based, the kernel prefetcher successfully predicts 65%, likely due to the compressed representation, which frequently places many child nodes on the same page. These applications leave limited room for improving prefetcher performance. In contrast, the kernel prefetcher predicts only about 8% of the pointer-based faults in Redis, indicating significant potential. These findings emphasize that identifying applications where the default kernel prefetcher is ineffective is a key part of designing a pointer prefetcher.

4 Design and Implementation

We begin by providing an overview of page fault handling in CheriBSD to illustrate how CHERI-picking fits into the existing code. Traditional prefetchers rely on memory access history, which is accessible at page fault time, whereas CHERI-picking relies on the *contents* of memory pages, which is available only after a page has been swapped in. This algorithmic difference produces a rather different implementation described in **??**.

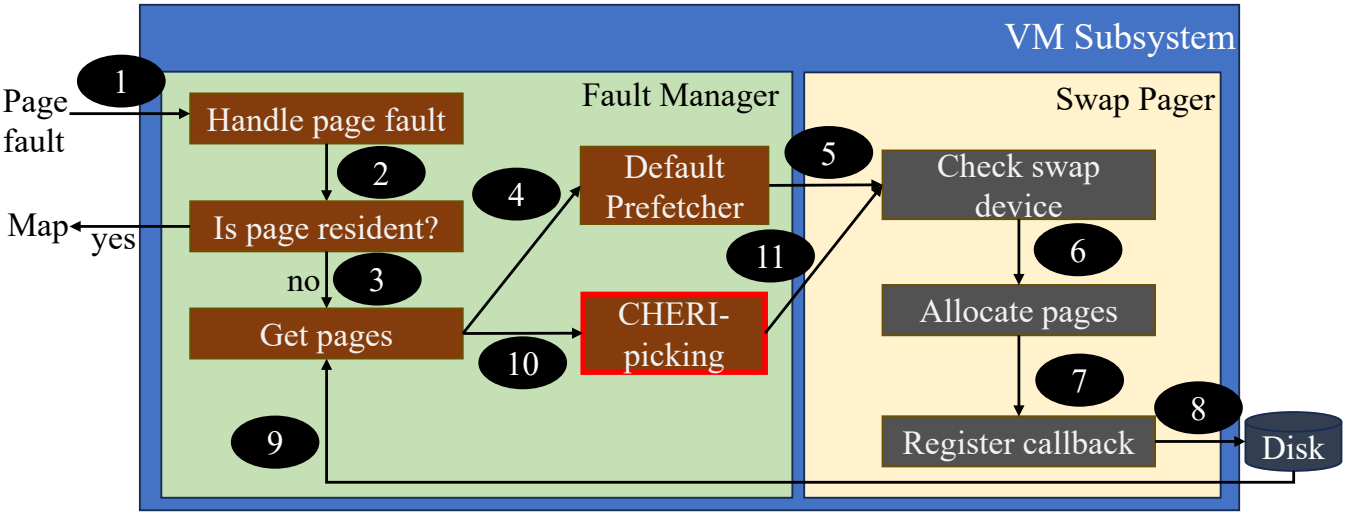

Figure 3. CheriBSD swap workflow. CHERI-picking is invoked after the I/O request is completed.

4.1 CheriBSD swap workflow

?? illustrates the high-level design of the CheriBSD swapping workflow; the numbers in this section refer to the figure. When a page fault occurs (#1), the operating system checks if the page is already in memory (#2). If it is, CheriBSD maps the page into the application's address space and resumes application execution; this is known as a **soft fault**.

If the page is not present in memory, the OS must retrieve the page from swap (in this case, the disk); this is called a **major fault**. The page fault dispatches a request to get pages (#3). While getting the pages, the kernel executes the default prefetcher to detect sequential accesses (#4). If the prefetcher detects a pattern, it checks if the requested page is present in the swap device (#5). If the requested page is in swap, it prepares for prefetching by allocating physical frames (#6). It registers a callback for prefetched pages to put them into appropriate page queues when their I/O requests are completed (#7). Then it finally issues an I/O request for the faulting page and any pages to be prefetched (#8). The swap manager returns after the faulting page is swapped in.

4.2 CHERI-picking policy

CHERI-picking is highly configurable. We begin with an overview of when CHERI-picking is invoked, a description of the algorithm, and then discuss some key parameters available for tuning and future research. CHERI-picking does not change the CheriBSD page fault path at all. Instead, after the callback for the faulting page occurs (#9); CHERI-picking runs (#10), and according to configuration parameters, prefetches some pages (#11)

?? describes the CHERI-picking algorithm. When the system swaps in a page, it obtains the kernel address for the physical page and iterates through its contents (lines 1-3). For each address, CHERI-picking queries the hardware to retrieve the tag bit to determine if the address contains a pointer (line 4). Upon finding a pointer, it confirms that the page is not already present in memory or prefetched (line 6-7). It then verifies if the corresponding page is present in swap (line 8). If the page is present in swap it is added to an asynchronous prefetch queue (line 9), and the swap manager fetches these pages into main memory. We prefetch a fixed number of pages, a parameter called *prefetch_count* (line 3), which can be tuned. We could limit ourselves to prefetching a small number of pages, pages whose pointers reside in certain ranges on the page, or addresses that meet any desired or learned criteria. CHERI-picking could also run on soft faults. We leave this kind of policy exploration for future work (**??**).

4.3 CHERI-picking design

We chose to implement CHERI-picking as a standalone function invoked by the callback to isolate our implementation during development and evaluation. This has several consequences and suggests directions for future work.

Rather than implementing CHERI-picking in the swap callback function, one could merge CHERI-picking with the CheriBSD function `swp_pager_meta_cheri_get_tags()` that restores a page's capabilities . While we have not yet done that, this will likely reduce CHERI-picking's runtime overhead penalty, as discussed in **??**.

Algorithm 1 CHERI-picking algorithm

Require: $faulting\_page, prefetch\_count$
$page\_addr \leftarrow kernel\_addr(faulting\_page)$
while $page\_addr < page\_addr + page\_size$ AND
    $count < prefetch\_count$ **do**
  $is\_ptr \leftarrow cheri\_gettag(*page\_addr)$
  if is_ptr **then**
    if $!page\_in\_ram()$ AND
      $!page\_prefetched()$ **then**
      if $page\_in\_swap()$ **then**
        $get\_page\_async()$
        $count++$
      end if
    end if
  end if
  $page\_addr += sizeof(CHERI\_Cap)$ ▷ 16 bytes
end while

Recall from **??** that CHERI's purecap compilation mode ensures that every pointer in an application is tagged by the hardware and treated as a capability. Thus, CHERI-picking works only on applications compiled in purecap mode.

As currently implemented, CHERI-picking always runs after the default prefetcher. On one hand, CHERI-picking detects, and prefetches accesses that the default prefetcher cannot. On the other hand, sometimes (as we'll see in the microbenchmark results in **??**), this can disrupt the performance of the default prefetcher. Better communication between the default prefetcher and CHERI-picking should be able to remedy this.

5 Evaluation

We use memory access traces of benchmark workloads to evaluate the performance of the proposed ML prefetchers. The metrics we will focus on is:

Definition 5.1. Coverage = $\frac{\text{Page faults predicted by Prefetching}}{\text{Total Page faults}}$

Definition 5.2. Accuracy = $\frac{\text{Page faults predicted by prefetching}}{\text{Total predictions}}$

We use LEAP as the baseline prefetcher for our evaluations to compare with the proposed ML prefetchers.

5.1 Data Collection and processing

To collecte page faults, we use `fltrace` , which will interpose on all **heap allocations**. Thus, we can collect page faults for all heap accesses. Stack accesses are excluded from the analysis since stack page faults are rare and are not the focus of this work.

Privous work has suggested that offsets of page address accesses works better than absolute addresses, and since we are doing classification, we need to filter out the rare classes and only leave up to 50,000 most occured classes for offsets and pc values.

Definition 5.3. Offset = Next Page Address − Last Page address

`fltrace` will need local RAM size to be set to run. It simulates the physical memory the program can access. Since each workload has a different memory requirement, we set the local RAM size to be 25% and 50% of the maximum memory requirement of the workload. This can be found by running `/usr/bin/time -v ./workload` and looking at the `Maximum resident set size`.

5.2 Workloads

We chose the following workloads for our evaluation:

- SPEC2017 : mcf omnetpp and lbm with the default input files offered by the benchmark suite.
- GAP : the default `bfs, pr, bc` workloads on the `twitter` dataset .
- WiredTiger : the default `ycsb-a` and `ycsb-c` workloads with `icount=30000000` in the benchmark configuration file.
- **Redis TO BE ADDED** : the default `redis-benchmark` workload with `-csv` option.

6 Challenges and Future work

Our focus in CHERI-picking was to demonstrate the feasibility of a generalized kernel pointer prefetcher. However, the prototype is not yet a complete implementation. While CHERI-picking improves coverage for BFS and canneal, it does not reduce the number of major faults for canneal; Canneal is a pointer-dense benchmark and a challenging one for CHERI-picking. Each faulted page contains many pointers; canneal first indexes into a pointer array randomly. If that access causes a page fault, CHERI-picking will prefetch 4 pages that will not be used. In fact, CHERI-picking prefetches 30 million pages, but only about 10% of those are ever accessed. BFS provides a nice contrast here; although it too is pointer dense, since we are traversing the entire data structure, every pointer prefetched is ultimately accessed. This results in 5% fewer major faults compared to the default prefetcher. Optimizing the CHERI-picking policy parameters such as `prefetch_count` is critical to reduce thrashing.

There are several avenues of performance optimization that we intend to investigate to address the overheads of CHERI-picking. First, we need to add better communication between the swap system, the default prefetcher, and CHERI-picking; by re-using swap data structures, we can optimize the CHERI-picking policy and record sufficient metadata to prevent CHERI-picking from running for cases when the default prefetcher is effective. Second, we do not run CHERI-picking on soft-faulting pages, but doing so asynchronously provides for more aggressive prefetching. Third, blindly prefetching a fixed number of pages on every faulted-in page is unlikely to be a good strategy; we need to explore how to better identify the right pointers to prefetch. Fourth, optimizing the I/O requests that CHERI-picking issues is essential; along with examining CHERI-picking performance on other swap backends such as RDMA. The overhead of scanning pages might be a bottleneck for faster swap backends such as CXL-attached memory or RDMA. To maintain performance in these scenarios, investigating techniques such as asynchronous execution of the CHERI-picking algorithm will be critical. Additionally, potential future hardware optimizations such as scanning the tags in hardware while copying a page can help remove the overhead.

7 Conclusion

CHERI-picking represents an initial step towards integrating application-specific prefetchers in the kernel. Our analysis reveals that both applications and benchmarks exhibit pointer-chasing patterns that the default kernel prefetcher fails to predict effectively. Through evaluation, we demonstrate benchmarks where the kernel prefetcher's effectiveness is limited, while CHERI-picking successfully predicts future accesses. CHERI-picking significantly enhances coverage for two standard benchmarks, improving it by up to a factor of three. However, much remains to be done to transform our prototype into a practical reality.

Acknowledgments

The authors would like to thank – Robert Watson for access to Morello hardware; George Neville-Neil, Jessica Clarke, Brooke Davis, John Baldwin, and Mark Johnston for their help with CheriBSD; Reto Achermann, Joel Nider, and the anonymous reviewers of PLOS for their feedback on the draft. We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC).

Temporary page!

LaTeX was unable to guess the total number of pages correctly. As there was some unprocessed data that should have been added to the final page this extra page has been added to receive it.

If you rerun the document (without altering it) this surplus page will go away, because LaTeX now knows how many pages to expect for this document.