CHERI-picking: Leveraging capability hardware for prefetching

Shaurya Patel
University of British Columbia

Sidharth Agrawal
University of British Columbia

Alexandra Fedorova
University of British Columbia

Margo Seltzer
University of British Columbia

Abstract

DRAM now accounts for over 30% of overall datacenter expense [30], due to its increasing cost and decreasing scaling [19, 22]. As applications demand more memory, operators look for cost-effective solutions to handle these increasing requirements.

One way to address the problem is to use disaggregated or far memory [18, 23, 25, 30]. Far memory solutions have an access latency approximately an order of magnitude slower than DRAM, thus, accurate memory page prefetching is critical. Important applications show pointer-chasing behavior, and existing prefetchers struggle to effectively predict these patterns. We find that 35-78% of page faults for benchmarks we analyzed are due to pointer accesses, but the default kernel prefetcher is ineffective for these patterns.

We introduce a new generalized kernel pointer prefetcher using CHERI: Capability Hardware Enhanced RISC Instructions [32]. Our approach, called CHERI-picking, leverages CHERI pointer capabilities to identify locations that contain pointers and prefetch the pages those pointers reference, subject to a policy. CHERI-picking does not require changes to applications, profiling, or offline analysis.

We implement CHERI-picking in CheriBSD and evaluate it using benchmarks. Our results show that CHERI-picking is effective where traditional kernel prefetchers are not, indicating the promise of this approach. We also show the overheads of discovering pointers and discuss blocking faults (faults that are prefetched but still in transit when the application accesses them) that currently stand in the way of adopting CHERI-picking. We discuss potential avenues to address these challenges.

ACM Reference Format:

1 Introduction

Increasing memory requirements for applications, e.g., machine learning, coupled with the slowdown of DRAM scaling [19, 22], makes DRAM one of the costliest components in data centers, constituting as much as 30% of the entire cost [30]. To accommodate increasing application memory demands without breaking the bank, operators often resort to far memory or memory disaggregation [3, 4, 13, 18, 25–27, 34]. Far memory incorporates additional tiers of slower memory, such as NVM, SSDs, or software-based approaches, e.g., compressed swap [2, 18, 30]. These tiers store memory pages that are less frequently accessed, freeing up costly DRAM for hot data.

There are various approaches to accessing far memory, such as application transparent approaches that use the swap subsystem [3, 4, 18, 27, 30] or accessing far memory from userspace [23, 25]. Regardless of the approach, accurate eviction and prefetching policies are essential to maintain application performance in the presence of far memory. Leap demonstrated that effective memory page prefetching in the kernel increases application performance by up to 10× [3]. Kernel prefetchers similar to Leap rely on information collected during page faults to accurately predict strides accessed [3, 9, 15]. However, these prefetchers are ineffective at predicting irregular memory access patterns.

Prior work reports that memory accesses in high-level languages are predominantly reference-based and irregular [27]. A reference or pointer-based pattern is common in pointer-chasing workloads, where the pointer’s value is crucial for predicting the memory access pattern. Pointer prefetchers have the ability to predict such patterns. For instance, during a linked-list traversal, a pointer prefetcher can prefetch the value of the next pointer. Recent systems introduced support in the kernel for application-level prefetching, with a specific focus on pointer prefetching [27, 34]. In both approaches, the kernel prefetches memory pages based on requests from application-level prefetchers. These systems produced up to 2× better performance in Redis [34] or 25% improvement in cache hits for Spark [27].
While prior work is encouraging, their implementations are limited to specific language runtimes [27] or specific applications [34]. Moreover, these approaches require that application developers implement non-trivial application-level prefetchers correctly. These approaches also require kernel support to allow prefetch requests from userspace. We address these shortcomings with a generalized kernel pointer prefetcher called CHERI-picking. CHERI-picking leverages hardware capabilities [32] in an application-agnostic manner. The key insight is that CHERI [32] treats all pointers as capabilities or fat pointers. The hardware manages the tag metadata that indicates which memory words contain pointers. The kernel can query the tag metadata to identify pointers stored in memory without application support. We design CHERI-picking using this insight and demonstrate how pointer prefetching can be implemented in the kernel and eventually replace application-specific implementations. Specifically, we address the following key questions:

- How can the availability of hardware capabilities in CHERI be used to design a pointer prefetcher; what are the challenges of using this approach?
- What fraction of page faults that applications experience are due to pointer accesses?
- What fraction of pointer page accesses does the default Linux prefetcher [15] successfully prefetch?

To understand how to use CHERI capabilities for prefetching, we present a prototype prefetcher implemented in CheriBSD [11].

By utilizing CHERI capabilities, CHERI-picking can identify pointers to make prefetching decisions. Similar to existing prefetchers, CHERI-picking runs at page fault time allowing it to consider spatio-temporal locality, as pointers on a currently faulted page are likely to be accessed in the near future. Unlike prior work, CHERI-picking is application-agnostic. To understand the applications that are suitable targets, we use dynamic analysis to analyze applications and quantify the potential of pointer prefetchers. We evaluate the CHERI-picking prototype using microbenchmarks and show that CHERI-picking can increase cache hits by 3.7×. But, using it for real workloads still remains challenging, due to its runtime overheads. We discuss potential avenues to address these overheads.

2 Background and Related work

2.1 CHERI

CHERI [28, 32] is an ISA extension that adds new architectural features to enable fine-grained memory protection. It complements the page-based access protection model that the MMU provides. Each pointer in CHERI is a 129-bit CHERI capability with a memory address in the lower 64 bits and other information, such as allowed permissions and range in the next higher 64 bits. To ensure that capabilities cannot be tampered with, the architecture also stores an additional tag bit (129th bit) in separate tag memory, which indicates that the memory address contains a capability. If an instruction tries to increase the bounds of the capability or extend its permissions, the tag bit is cleared, thus invalidating the capability. When a memory address is accessed via capability, the CPU allows access only if the capability is valid and has the right permissions. In “purecap” compilation mode [29], every pointer of an application is treated as a capability, giving the hardware fine-grained information about what memory ranges the application can access. The CHERI instruction set also provides an instruction to read the tag bit, which makes it possible to identify pointers stored in memory. This instruction was used by Cornucopia [31] and CHERIvoke [33] to prevent temporal memory bugs (such as use after free) by scanning the process heap for pointers, i.e., capabilities, at runtime. In contrast, we use this instruction to detect addresses to prefetch.

CheriBSD [11] is a port of the FreeBSD operating system to CHERI-enabled hardware. For an operating system to run on CHERI and to provide CHERI capabilities to its userspace, the port had to change exception handling, process loading, and the swap subsystem. We focus only on the swap subsystem modifications made in CheriBSD. When a memory page is swapped out, it might contain CHERI capabilities (pointers), so CheriBSD saves those tag bits. It then restores those tag bits when the page is swapped back in. This added overhead due to saving and restoring the tag bits is an unavoidable overhead of using CHERI.

2.2 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 [9, 15] and are successful at fetching spatially related pages. Leap [3] improved traditional prefetching using majority trend detection to identify striped 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. Memliner [26] coordinates memory accesses from the garbage collector (GC) and an application such that GC accesses don’t interfere with the application’s access history. This technique increases the efficiency of Leap. We focus on prefetching pointer-chasing patterns, which are usually irregular.

Prior work [27, 34] addressed Leap’s shortcomings to varying degrees. Canvas [27] introduced a new mechanism to 1) isolate the swap subsystem and the memory access histories of threads inside the kernel and 2) issue an upcall to the application-specific prefetcher if the default prefetcher [15] cannot decide which pages to prefetch. The prefetcher runs in the JVM for Java applications and can prefetch pointer-chasing patterns. We focus on prefetching pointers in the
general case without changing the language runtime. Di-LOS [34] introduced a libOS-based approach to disaggregated memory that allows the system to perform application-level prefetching without the overhead of upcalls. Using an application-level prefetcher for Redis, DiLOS improves the performance of the kernel prefetcher in prefetching pointer-chasing workloads. CHERI-picking focuses on generalizing such application level approaches.

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 [15]. Prior work performed a similar analysis to determine whether pointer chasing causes cache misses [14]. 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 [14] 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 Listing 1. The 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.

```
curr = curr->next;
mov -0x600028(xrbp),%rax; load address of curr to rax
mov 0xff0(%rax),%rax; load the value of curr->next
mov %rax,-0x600028(xrbp); 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) [21] and the page fault trace using perf [1] 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.

<table>
<thead>
<tr>
<th>Workload</th>
<th>Source</th>
<th>WSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array streaming</td>
<td>Microbenchmark</td>
<td>50%</td>
</tr>
<tr>
<td>Random linked list traversal (LL)</td>
<td>Microbenchmark</td>
<td>50%</td>
</tr>
<tr>
<td>Canneal</td>
<td>Parsec [8]</td>
<td>50%</td>
</tr>
<tr>
<td>xHPCG</td>
<td>HPCG [12]</td>
<td>50%</td>
</tr>
<tr>
<td>Redis benchmark LIRANGE</td>
<td>Redis [24]</td>
<td>25%</td>
</tr>
</tbody>
</table>

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

![Graph](image1.png)  
**Figure 1.** Percentage of pointer-based page faults for different workloads

![Graph](image2.png)  
**Figure 2.** Accuracy of the kernel prefetcher on pointer-based page faults.
3.3 Pointer-based page faults

We analyzed different benchmarks (Tab. 1). 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 [6, 34] having identified that they would benefit from pointer prefetching: Parsec’s canneal [8], xHPCG [12] and Redis [24]. 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 [20].

Fig. 1 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 2 illustrates the accuracy of the default kernel prefetcher [15] 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 [7] 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 [11] 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 Sec. 4.2.

4.1 CheriBSD swap workflow

Fig. 3 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).

Alg. 1 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 (Sec. 6).

**Algorithm 1 CHERI-picking algorithm**

```
Require: faulting_page, prefetch_count
1:   page_addr ← kernel_addr(faulting_page)
2:   while page_addr < page_addr + page_size AND
3:       count < prefetch_count do
4:       is_ptr ← cheri_gettag(*page_addr)
5:       if is_ptr then
6:           if !page_in_ram() AND
7:             !page_prefetched() then
8:               if page_in_swap() then
9:                 get_page_async()
10:                count++
11:           end if
12:        end if
13:     end if
14:     page_addr += sizeof(CHERI_Cap) > 16 bytes
15:   end while
```

**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 `swap_page_meta_cheri_get_tags()` that restores a page’s capabilities [10]. While we have not yet done that, this will likely reduce CHERI-picking’s runtime overhead penalty, as discussed in Sec. 5.3.

Recall from Sec. 2 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 Sec. 5.1), 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 assess CHERI-picking’s performance on key prefetching metrics by comparing it to the default kernel prefetcher [9]. We use a subset of the workloads we analyzed in Sec. 3. Specifically, we evaluate: the linked list microbenchmark, canneal on the native dataset, and BFS on the Wikipedia-links dataset [16]. We do not further analyze xHPCG as we saw that it is not pointer dense and has relatively few page faults (~50k). We also don’t analyze the array traversal benchmark as the default prefetcher is expected to be effective. We leave further analysis of Redis as future work.

We implemented CHERI-picking in the CheriBSD kernel version 22.12 [11], which delays mapping prefetched pages until they are accessed, unlike the default CheriBSD kernel, which proactively maps in prefetched pages. We run evaluations on an ARM Morello Cheri-capable processor [5] that contains 4 cores running at 2.4GHz. We limit memory so that the working set size of applications is twice that of the available memory, inducing memory pressure.

We evaluate prefetching performance using the following metrics:

- **Soft faults**: These page faults occur when a page is already in memory, but not mapped into an application’s address space; indicating the prefetcher’s prediction capacity.
- **Major faults**: These page faults occur when the page is not present in memory and indicate the faults not predicted by the prefetcher, encompassing mandatory misses and prefetcher miss predictions.
- **Coverage**: The percentage of page faults that were satisfied by previously prefetched pages.

In Sec. 6, we discuss CHERI-picking performance overhead and challenges.
### 5.1 Microbenchmark results

We use two different versions of the linked list microbenchmark from Sec. 3. In one version, we traverse the list in allocation order, while in the other, we traverse the elements in random order. We allocate a total of 500k pages (equal to 2GB). When we order the pages in allocation order (sequential), the default prefetcher proves effective, as demonstrated in Tab. 2. The default prefetcher’s initial prediction requires two sequential major faults to detect a sequential pattern. However, CHERI-picking, which also prefetches pages here due to pointer-chasing, eliminates the second major fault and prevents the default prefetcher from detecting the sequential pattern. Further, CHERI-picking is worse than the default prefetcher here, because it prefetches one page at a time (in the current prototype), while the default prefetcher could fetch several (minimum 7 pages). This interference issue could be resolved by combining the two prefetching algorithms or communicating between them.

When we order the pages randomly, CHERI-picking works as expected and outperforms the default prefetcher; as we saw above, it still prefetches only 50% of the faulting pages, due to its not running at soft fault time. Given the random order of page accesses, the default prefetcher fails to predict any of the faults. This scenario can be present in various pointer-chasing workloads where the data structure’s allocation and access order differ.

<table>
<thead>
<tr>
<th>Workload</th>
<th>Soft faults (higher is better)</th>
<th>Major faults (lower is better)</th>
<th>Coverage (higher is better)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Default</td>
<td>CP</td>
<td>Change</td>
</tr>
<tr>
<td>Linked list sequential</td>
<td>401K</td>
<td>227K</td>
<td>0.56×</td>
</tr>
<tr>
<td>Linked list random</td>
<td>11K</td>
<td>236K</td>
<td>21.45×</td>
</tr>
<tr>
<td>canneal</td>
<td>953K</td>
<td>3543K</td>
<td>3.7×</td>
</tr>
<tr>
<td>BFS</td>
<td>149K</td>
<td>204K</td>
<td>1.35×</td>
</tr>
</tbody>
</table>

Table 2. CHERI-picking (CP) improves the number of soft faults on standard benchmarks by up to 3.7×. Values in red indicate instances where CP performs worse than the default prefetcher.

However, CHERI-picking can predict the pointers accessed by the two lists inside the element, leading to an increase in soft faults and coverage compared to the default prefetcher.

The default prefetcher worked well for BFS, based on the results from Fig. 2. CHERI-picking improves coverage by 13% over the default prefetcher, demonstrating that pointer prefetchers can improve coverage for BFS, albeit only by a small amount.

### 5.3 CHERI-picking overhead

Currently, CHERI-picking has two major sources of overhead. First, when we traverse the page looking for pointers, this adds latency to the page fault. Second, when we don’t prefetch the page early enough (timeliness), it leads to blocking soft faults. The execution time of the sequential linked list microbenchmark with CHERI-picking is twice that of the default prefetcher because CHERI-picking adds about 7µs to every page fault (which is clearly unacceptable) and also disrupts the default kernel prefetcher (Sec. 5.1). The execution time of the random linked list microbenchmark is the same for both prefetchers. Even though this is a pointer-chasing workload, we do not see a performance boost, because CHERI-picking also causes many blocking soft faults, which occur when a prefetched page is still in transit from disk. In part, this is due to the nature of our microbenchmark; we do not process the nodes on the list at all. In reality, there will be some processing on each node in the list, which could mitigate these blocking faults. In canneal, for example, only 13% of soft faults block.

### 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.
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 where 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).

References
