Eden: Developer-Friendly Application-Integrated Far Memory

Anil Yelam, *UC San Diego*
Stewart Grant, *UC San Diego*
Saarth Deshpande, *UC San Diego*
Nadav Amit, *Technion, Israel Institute of Technology*
Radhika Niranjan Mysore, *VMware Research*
Amy Ousterhout, *UC San Diego*
Marcos K. Aguilera, *VMware Research*
Alex C. Snoeren, *UC San Diego*

Abstract

Far memory systems are a promising way to address the resource stranding problem in datacenters. Far memory systems broadly fall into two categories. On one hand, paging-based systems use hardware guards at the granularity of pages to intercept remote accesses, which require no application changes but incur significant faulting overhead. On the other hand, app-integrated systems use software guards on data objects and apply application-specific optimizations to avoid faulting overheads, but these systems require significant application redesign and/or suffer from overhead on local accesses. We propose Eden, a new approach to far memory that combines hardware guards with a small number of software guards in the form of programmer annotations, to achieve performance similar to app-integrated systems with minimal developer effort. Eden is based on the insight that applications generate most of their page faults at a small number of code locations, and those locations are easy to find programmatically. By adding hints to such locations, Eden can notify the pager about upcoming memory accesses to customize read-ahead and memory reclamation. We show that Eden achieves 19.4–178% higher performance than Fastswap for memory-intensive applications including DataFrame and memcached. Eden achieves performance comparable to AIFM with almost 100× fewer code changes.

1 Introduction

As the memory requirements of datacenter applications increase without a commensurate decrease in DRAM costs, datacenter operators are increasingly concerned by memory stranding. At the same time, host interconnects are delivering higher bandwidth and lower latency [19, 24, 26, 27, 29, 35]. As a result, there is a renewed interest in far-memory systems which enable servers to access under-utilized DRAM on other servers or pooled on dedicated devices within a rack or cluster. Recent efforts have explored a range of issues that arise including prefetching, access models, and fault tolerance [8, 13, 15, 23, 25, 38, 41, 42, 44, 49, 52, 53] but they all make sacrifices along at least one dimension: performance, developer effort, or flexibility.

Existing far-memory systems generally take one of two approaches. Paging-based systems like Infiniswap [23], Fastswap [13] and their extensions [12, 38, 49] rely on *hardware guards* to intercept accesses to remote memory: the microprocessor checks if a page is not locally present and in such case triggers a page fault to fetch it remotely [9]. This approach works with unmodified applications and incurs little overhead on local accesses, but suffers from two issues. First, page faults are expensive because they require trapping into the kernel—which takes on the order of a microsecond, making it impractical to context switch to a different task while waiting for a page fetch to complete (which takes 5–6 μs in our RDMA cluster). As a result, systems such as Fastswap busy wait on page fetches, wasting CPU cycles and limiting performance [13]. Second, their transparent page-based approach does not provide the kernel with the information needed to implement application-specific policies to improve performance (e.g., custom prefetching or eviction).

Alternatively, app-integrated systems such as AIFM [41] and Carbink [53] manage memory in user space at object granularity and rely on *software guards* to intercept remote accesses: applications use remotable pointers which, when dereferenced, check if the data is remote and if so fetch it. Their user-space implementation avoids the overheads of kernel-moderated page faults and makes use of the CPU during a remote fetch by switching to a different lightweight user-level thread. In addition, by operating at object granularity, they can expose data-structure-specific access patterns to the runtime to enable application-specific optimizations. However, app-integrated systems must pay the cost of software guards even on local accesses, and require significant developer effort to port applications to use remotable pointers. Systems like TrackFM [44] and Mira [25] employ compiler techniques to decrease the porting effort but still incur software guard overheads and require source code access to all of the library dependencies of the application, which may be infeasible in many scenarios (§2.1).

We propose Eden, a page-based system that combines software and hardware guards to expose memory access patterns without imposing a disruptive object-based interface. Eden is based on the insight that far-memory applications tend to generate their page faults at a small number of code locations, and those locations are easy to find programmatically. We provide a tool to identify fault locations, and our study of 22 applications shows that, at the median, 12 code loca-

tions cover 95% of page faults. Hence, Eden deploys software guards (code annotations) at these few performance-critical code locations to presage most remote accesses, while relying on hardware guards (page faults) to service the few remaining remote accesses that may occur elsewhere in the application.

The parsimonious use of software guards balances their expressive power against their runtime overhead. By capturing most far memory accesses with code annotations, we avoid almost all expensive kernel-mediated page fetches. Moreover, we show that Eden can effectively coalesce related far memory accesses to provide the same type of application-specific optimizations as systems that resort to object-based interfaces, but with significantly less programmer effort. Finally, by employing software guards only where accesses are likely to be remote, Eden minimizes their overhead.

Based on this insight, Eden asks the developer to instrument these code locations with memory-access hints. By using a hybrid kernel/user-space page management scheme, these hints allow Eden to tailor memory policies to the application (including fetch, eviction, and read-ahead policies) and to leverage a lightweight scheduling framework to run user-level threads during page fetches, thereby improving performance. Specifically, a hint transfers control to Eden's user-space runtime, which checks for page presence and detects an impending fault without trapping into the kernel. If a page is missing, Eden initiates the fetch from user space.

Efficient joint kernel/user-space page management is challenging to realize, as some parts of page-fault handling must occur in kernel context. For example reading the faulting address is a privileged operation on x86 processors. Similarly, mapping and unmapping pages must occur in kernel space. Finally, hints may not be perfect, so Eden's user-space runtime may not catch every fault. Thus, employing a page-based approach with today's CPUs fundamentally requires some kernel participation, whose overheads, if left unchecked, can quickly overwhelm any potential performance benefits.

Eden uses two key ideas to overcome the challenges of hybrid kernel/user-space page management. First, because the kernel retains control of page tables in Eden, syscalls are still needed for tasks such as mapping pages after they are fetched, toggling write protection, and indicating when pages can be reclaimed from a process' address space. Our design streamlines these operations by extending Linux's `userfaultfd` and `madvise()` syscalls to provide *vectorized APIs*. Using these enhanced syscalls, Eden amortizes the cost of necessary kernel traps across many pages, dramatically improving performance.

Second, Eden further uses its hints to pass additional application-specific information to guide prefetching and reclamation. As the hinted accesses cover most page faults in the application, these hints provide a great opportunity to pass data-access knowledge gained either through developer insight or offline profiling. For example, in loops or when accessing large objects, hints can provide precise read-aheads to batch page faults and amortize the page-in cost. Similarly, faulting accesses to certain data structures can be prioritized for reclaim over others by setting different priorities for different hints. We show that such *extended* hints help Eden perform on-par with app-integrated approaches that require significant developer effort. While Eden relies on developer insight to extend hints (similar to app-integrated systems), the effort is smaller and we carefully design the hint extensions to be simple enough for automatic detection and injection based on offline profiling.

We implement Eden on top of Shenango [36], a highly scalable user-level thread scheduling runtime, which Eden leverages to efficiently switch from one thread to another while a hinted fault is being resolved. We evaluate Eden across a variety of workloads both from a performance and developer-effort standpoint. We show that Eden achieves higher throughput than a recent kernel-based far memory system, Fastswap [13], by 19.4–178% on real-world applications such as DataFrame and memcached. Furthermore, Eden achieves similar throughput to an app-integrated approach, AIFM [41], on both DataFrame and synthetic Web service workloads with sufficient local memory. In terms of developer effort, we show that for several example applications, only a handful of hints (2–32) are required to handle 95% of page faults in Eden. In contrast, the AIFM authors modified 1,192 lines in the DataFrame library—108× more than Eden. Our code is openly available at `https://github.com/eden-farmem/eden`.

2 Background and Motivation

In this section we justify our key design decisions. First, we explain the issues with prior app-integrated systems that we address with our hybrid approach. We then document the overheads inherent in managing traditional, page-based virtual memory from user level and suggest how hints can be used to both communicate application-specific workload information and ameliorate the cost of user/kernel transitions. And finally, we show that there are typically only a small number of locations where such hints are needed so introducing them requires limited developer effort.

2.1 Issues with prior app-integrated systems

The original app-integrated approach to far memory, used in AIFM [41] and Carbink [53], entails significant developer effort and performance overhead due to its remotable pointers. To use these systems, the developer must identify every location where a remote pointer dereference might occur and insert a software guard at each. The guard checks that the data is in local memory, and if not, reads it from far memory. Adding these guards can involve significant modifications; for example, porting the DataFrame library to use AIFM required modifying 7.7% of the lines of code.[1] In addition, while these

[1]The AIFM authors reported modifying 1,192 lines of code [41] out of 15,525 lines of C, C++, and makefiles in the library [2].

guards are highly optimized, they still incur overhead, even if the check succeeds (i.e., the data is already in local memory).

For example, AIFM's guard contains 5 instructions in the best case, and can take up to 489 cycles at the 90th percentile even when data is in local memory. (In contrast, the hardware guards of paging-based approaches use highly optimized TLB hardware.) Unfortunately, for functional correctness, guards are needed throughout the application (anywhere a pointer to the heap may be dereferenced) resulting in a large number of guards. To avoid invoking guards on every access, AIFM employs programmer-assisted *dereference scopes*—small blocks in the program code—wherein the object is marked unevictable and guards are avoided using native pointers. Such optimizations, while helpful, are not always applicable (e.g., streaming workloads where objects experience a single access) and may even add to the programmer burden. For example, the DataFrame library above sees up to 30% slowdown with AIFM even after these guard optimizations. Eden does not need such optimizations because it uses hardware guards where accesses are likely to be local.

More recently, systems such as TrackFM [44] and Mira [25] use compiler techniques to programmatically insert software guards. While effective at reducing programmer burden, neither can eliminate the overhead of guards (e.g., TrackFM's guard contains 14 instructions). Both TrackFM and Mira try to reduce the number of guards required through static analysis, but such techniques have limits (e.g., when the compiler cannot statically resolve branching, function calls, or shared accesses from multiple threads). Ultimately, even with optimizations, TrackFM's overheads are significant: up to 40% slowdown in application performance (§5). Furthermore, for correctness, both systems require instrumenting most or all external library dependencies of an application, which may not be possible (e.g., when source code is not available). In addition, the instrumentation produces larger binaries, incurs higher compilation times—TrackFM reports 2.4× bigger binaries and 6× longer compilation times on average—and complicates the tasks of debugging and profiling.

2.2 Page management overheads

When we combine software and hardware guards, any piece of remote data may be fetched through either our user library (software guards) or page faults (hardware guards). Our library thus needs a way to track and manage page faults to ensure consistent operation: if the user library fetches a page, it should be marked as present in its page table entry, so that subsequent accesses do not cause a page fault; similarly, if the fault handler fetches a page, our software guard should know that the page is local.

The Linux kernel has a simple mechanism to manage page faults in user space called `userfaultfd` [4]. To use `userfaultfd`, a process first registers a region of virtual memory with the kernel and receives a file descriptor in return. Then, if the process faults on an address in that region, the kernel notifies the process by sending an event to the file descriptor. Typically, the process will dedicate a core to polling the file descriptor to detect page faults and resolve them by using the `UFFDIO_COPY` `ioctl()` to map the missing page in the process' address space and unblock the faulting thread. When pages are no longer needed, processes use the `madvise()` syscall to ask the kernel to unmap them, and processes use another `ioctl()`, `UFFDIO_WRITEPROTECT`, to write-protect dirty pages while they are written out.

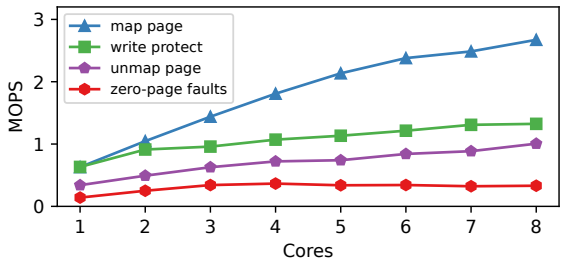

Figure 1: The maximum throughput of memory operations when performed from user space on Linux, as a function of the number of cores. Faults are serviced by four dedicated handler cores; additional cores do not improve performance.

However, we find that `userfaultfd` can entail significant overheads. Figure 1 presents a simple microbenchmark in which a variable number of application cores trigger each of the above memory-management operations as quickly as possible. (We service the zero-page faults on separate handler cores; the other three operations are performed locally at each core.) Even when faults are resolved in the simplest way possible—supplying a zeroed page with `UFFD_ZERO`—we find that the maximum rate at which (any number of) handler cores can poll for faults via a single `userfaultfd` file descriptor is 350 thousand faults per second on our hardware ("zero-page faults" plotted in red).[2] A far-memory system that handles all page faults via `userfaultfd` would only be able to handle ∼12% percent of the 3 million page transfers per second that are possible with a 100-Gbps NIC. While the other three operations are more lightweight, both unmapping (shown in purple) and write protecting pages (green) scale poorly due to the need to flush TLBs across all impacted cores.

Fortunately, our approach invokes this expensive `userfaultfd` machinery only occasionally, as hints can invoke our user-space page-management code before a page fault occurs. Moreover, we provide some simple vectorized enhancements to system calls to reduce the overheads of context switching to the kernel; with these enhancements, our library can manage multiple pages with a single system call.

2.3 Faulting locations

We validate our intuition regarding the number of different places in an application's source code that generate page

[2]Using additional file descriptors can improve throughput, but requires sharding far memory into one segment per `fd` (and, hence, across multiple handler cores) potentially causing load imbalance across handler cores.

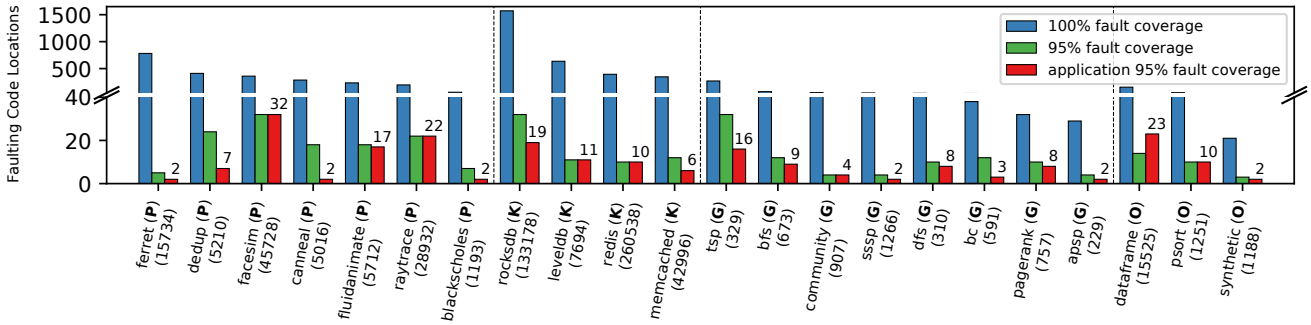

Figure 2: The number of unique lines of source code that account for 100% (blue) and 95% (green, localized to application code in red) of the page faults in 22 different applications. Local memory is configured to hold only 10% of the working set. The *x*-axis labels indicate the total number of lines of code in each application and the application's category (bolded letter).

faults in a far-memory context—and, therefore, require a programmer to add hints—by conducting a study across a set of 22 applications drawn from different benchmark suites to cover a range of application categories. (These results were previously published in a workshop paper [51]; we reproduce the high-level findings here for context.)

Specifically, we consider HPC applications from **P**ARSEC [14], **K**ey-value stores from LK_PROFILE [47] (e.g., LevelDB [5] and Memcached [32]), **G**raph algorithms from CRONO [11], and three **O**ther applications used in our evaluation. We run each application with its standard benchmarking workload to ensure realistic code coverage. Details about the applications and their workloads are provided as Supplementary Material (§A). Using our Eden-based fault-tracing tool (§4), we observe the behavior of each application while restricting the amount of local memory available to 10% of its maximum resident set. The tool identifies all unique fault-triggering source code locations in the application and the libraries it uses.

Figure 2 shows for each application both the total number of code locations that trigger faults and the number of code locations responsible for 95% of faults. In most cases, a small number of locations cover 95% of faults—12 locations at the median and fewer than 32 for all applications. Moreover, when we trace the faults back to the line of application code that led to the fault, even fewer unique lines of code are implicated: less than 10 in most cases. Further experimentation on these and other applications published elsewhere [51] shows that the set of locations is robust across different degrees of memory pressure. Manual analysis reveals the reason is straightforward: few code paths dominate runtime in these applications. At even 10% of the maximum resident set size, local memory is sufficiently large that each page brought in on a code path tends to remain available for later page accesses in that path. As a result, only the initial reference on a given path is likely to cause a fault.

3 Eden Design

Eden is a far memory system that aims to achieve three goals:

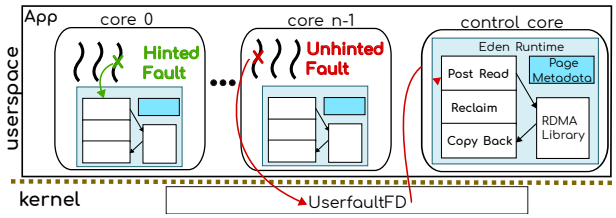

Figure 3: Eden's runtime runs on each application core and a dedicated control core. Hinted faults are handled directly by the runtime on the faulting core while unhinted faults are caught by the kernel and steered to the control core.

1. **Low developer effort:** developers should not have to extensively modify their application to use Eden.
2. **Flexible policies:** applications should be able to specify prefetch and eviction policies, and to leverage application-specific information in these policies.
3. **High throughput:** Eden should minimize its overheads as much as possible to deliver high throughput.

Eden's key idea is to manage remote memory accesses with software guards in the common case while avoiding these guards for most local accesses. It achieves this by having the developer add (a small number of) hints to their program about locations where remote accesses are most likely to occur. Eden's runtime can then efficiently check in user space whether the needed page is present, and initiate its fetch if not. In addition, Eden's hints provide an opportunity for applications to convey additional application-specific information about their memory-access patterns to customize prefetch and reclamation policies. Finally, for operations like write-protecting and unmapping pages which require kernel privileges, Eden provides low-overhead vectorized syscalls.

Figure 3 shows Eden's overall design. Application cores run applications in lightweight, user-level threads. Eden's runtime runs on each core and provides general-purpose functionality such as threading and networking. In addition, the runtime implements memory-management functionality such as logic for handling hints, fetching and evicting pages, and

```
hint_fault(
  /* basic hint */
  address, /* faulting address or page */
  mut = false, /* map read-only or writable */
  size = PAGE_SIZE, /* size of the region */
  /* extended hint */
  rdahead = 0, /* positive or negative read-ahead */
  ev_prio = 0, /* eviction priority */
  seq = false, /* sequential access */ );
```

Figure 4: Eden's hint API. Basic hints suffice to signal potential impending page faults; developers wishing to convey additional workload information can use extended hints.

prefetch and reclaim policies. In addition to the application cores, Eden's design includes a dedicated *control core* which is responsible for proactive memory reclamation and handling any "unhinted faults", i.e., those remote accesses which are not presaged by Eden's hints and therefore trigger a hardware guard (i.e., page fault). The kernel steers these (relatively rare) faults to the control core using `userfaultfd`.

3.1 Hint API

Hints are annotations that a developer adds to their program to notify the runtime of a potential page fault and to convey application-specific information about memory access patterns. The runtime uses the information provided by hints to fetch missing pages and avoid almost all page faults, as well as to improve its policies for prefetch and memory reclaim.

3.1.1 Basic hints

Figure 4 shows Eden's hint API. It includes two types of hints: basic hints and extended hints. Eden expects applications to provide at least the basic hints; otherwise all page faults will be handled with high overhead via traditional hardware guards and `userfaultfd`. The key piece of information that each hint conveys is the upcoming memory address (or region) that, if unhinted, might trigger page fault(s), and whether the faulting pages should be mapped read-only or writable. These addresses need not be page aligned. Eden strives to make placing hints as easy as possible, both in terms of lines of code modified and time spent to identify hint locations. We provide a tool to assist in this process (§4).

3.1.2 Extended hints

Eden's extended hints allow motivated developers to pass application-specific knowledge to the runtime. For example, an application can trigger prefetching by using `rdahead` to specify a number of batched pages to read either immediately before or after the hinted address. When the `size` of the hinted region is large, read-ahead can be extremely useful as the entire region can be fetched in a few `rdahead`-size chunks, amortizing the cost of page faults. Note that unlike `size`, `rdahead` is only suggestive and does not necessitate fetching all the pages in the window (e.g., when necessary page locks cannot be obtained). Eden also lets the application specify a page's eviction priority with `ev_prio`, which enables cache separation similar to AIFM's non-temporal hints. In addition, developers can use the `seq` flag to reduce the overhead of hints for workloads that scan data sequentially (e.g., array lookup in a `for` loop). When the `seq` flag is set, the runtime saves the last page accessed by a hint and skips the hint-handling process if the next invocation of that hint points to the same page.

Extended hints enable Eden to convey far more information than what is available to existing prefetch and eviction policies that run inside the kernel. Existing prefetching policies in Linux tend to rely on detecting sequential accesses or using more sophisticated algorithms to detect more complex access patterns [12]. In contrast, when an application knows (even parts of) its memory-access pattern, it can use extended hints to directly convey this information to Eden. The API could be extended to convey access patterns as well (for example to specify non-contiguous pages to prefetch) but we defer a full exploration to future work, focusing instead on leveraging deliberately limited annotation.

3.2 Handling page faults

In Eden, there are two paths for handling page faults, depending on whether the remote access is detected by a software or hardware guard. The *hinted path* corresponds to instances where the address is signaled by a hint before the program attempts to access it—thus, the fault is detected by Eden's software guards and no trap to the kernel is required. The *unhinted path* is followed if the application actually traps to the kernel due to a hardware guard, i.e., traditional page fault.

In both cases, if the page is not present, Eden's runtime fetches the page—or pages, in the case of extended hints—from far memory via RDMA and uses the `UFFDIO_COPY ioctl` to allocate a physical page, copy the fetched data in, and map it in the process' page tables. If the fault is due to missing write privileges, the runtime uses a `UFFDIO_WRITEPROTECT ioctl` to remove write-protection on the page. (The expensive `ioctl` calls are necessary in both hinted and unhinted paths because, unlike pure app-integrated systems, Eden needs to support hardware guards as well. While Eden is in control of what pages are mapped locally, the kernel still controls the page tables and translation hardware that support the hardware guards. Hence, when evicting pages, Eden must unmap them from the page tables.) However, the two paths differ in several aspects including which core handles the fault and whether detecting the fault involves context switching to the kernel.

Hinted path. Eden expects to handle the majority of page faults via the hinted path. When an application invokes Eden's hint API (i.e., `hint_fault(...)`), it calls into Eden's runtime, which implements a software guard. The runtime computes the page-aligned address and looks up the page in its own internal data structure of page metadata. If the page requires that write-protection be removed, the runtime issues a synchronous `ioctl()`. Otherwise, if the page is present, the

runtime returns to the application immediately. This process completes entirely on the same core as the application.

If the page is missing, the runtime fetches it from far memory and runs another user-level thread while the page is being fetched (unless the hint specifies otherwise). First, the runtime marks the current user-level thread as blocked. Then it initiates the RDMA read for the page. While the page is being fetched, Eden's runtime can switch to a different user-level thread and run it. This enables Eden to effectively utilize CPU cycles while a page fetch is outstanding, in contrast with existing systems such as Fastswap that busy wait [13]. When control returns to the runtime (e.g., the thread yields, exits, or issues another hint), it polls for any completed RDMA operations on its local queue. On a page-fetch completion, the runtime `UFFDIO_COPY`s the page and resumes the original user-level thread. While issuing the `ioctl()` still requires a context switch to the kernel, hinted faults only require one such context switch, whereas unhinted faults (described below) require two. In addition, the copy happens on the same core, warming up the cache for the application thread, unlike in the unhinted path.

Unhinted path. Hints may not catch all page faults in Eden, so regular page faults can still occur. This can happen if a developer instruments most of—but not quite all of—the potential fault locations, or the system experiences unexpected memory pressure. Eden handles these faults in the control core. In Eden, each application's memory is registered with a single `userfaultfd` file descriptor (§4). When an unhinted fault occurs, traditional hardware guards detect the fault and trap to the kernel, which then writes an event to the `userfaultfd` file descriptor. The control core polls this descriptor and receives the event. It resolves the fault in manner similar to the hinted path with a key difference: Because the control core does not run application threads, it handles other faults or memory reclaim (§3.3) while a page fetch is outstanding. When unhinted faults are rare, the control core is not a bottleneck.

Concurrent faults. It is possible that while a fault (either hinted or unhinted) is being resolved, another user-level thread faults on the same page. The runtime must not re-execute the fault-handling logic for the second fault, because this would trigger an extra page fetch and could potentially overwrite the page of memory after the first thread had already resumed. Eden handles this by including a lock in its page metadata, which the runtime acquires when it starts working on a page fault to ensure concurrent faults are resolved exactly once. When the runtime encounters a concurrent fault, it places the fault on a local waiting queue and proceeds to do other work. When the first fault finishes, the runtime sets the page status bits in global page metadata. Whenever a core returns to the runtime, it checks for completed faults and releases the blocked threads.

Fault stealing. In the common case, Eden resolves hinted faults on the core that triggered them. However, when load is imbalanced or deadlocks occur, Eden can also resolve faults on a different core via work stealing. Load can become imbalanced if an application thread runs for a long time without yielding (Eden inherits Shenango's non-preemptive cooperative threading model [36]) while another core sits idle. Deadlocks can arise due to asynchronous page fault handling where application cores block while waiting for locked pages to be handled by other cores that in turn wait on the former cores. Eden addresses both of these challenges with fault stealing. Faults can be stolen (served) not just by other application cores, but by the runtime on the control core. This is necessary for correctness as all the application cores may block. We ensure progress because the control core never blocks, so any faults waiting past a timeout get handled.

3.3 Memory reclamation

In Eden, both the control core and application cores participate in memory reclaim; this is similar to Fastswap [13], which has a dedicated "reclaimer core". Reclaim involves evicting enough cold pages to maintain the fraction of empty page frames at a configurable threshold. (We use 1% by default.) Most memory reclamation in Eden happens on the control core, since only a small portion of its time is consumed with handling unhinted faults. In some instances, however, the control core may become overwhelmed. In this case (i.e., when there are no empty frames), a faulting core will perform its own reclamation (i.e., instead of running another user-level thread while a page fetch is outstanding).

Regardless of which core performs reclamation, the runtime first must identify which page(s) to reclaim by running a reclaim policy (§4). Once the runtime has selected a page to evict, it first locks the page to check whether it is dirty. (Eden initially write protects pages pulled in by read faults and sets the dirty bit on a write hint or a `userfaultfd` write notification.) If the page is dirty, Eden write protects it using `UFFDIO_WRITEPROTECT` so that it cannot be modified during eviction. Then it copies the page to a buffer and writes it out to far memory using RDMA. Once this is complete—or if the original page was clean to begin with—the runtime removes the page from process memory with the `madvise(MADV_DONTNEED)` syscall.

3.4 Vectorized syscalls

The Userfaultfd syscalls that Eden relies on for memory reclamation (specifically, for write-protecting and unmapping pages), scale poorly due to the need to flush TLBs across cores (§2.2). These operations limit the rate at which Eden can possibly evict pages, and therefore the rate at which Eden can fetch pages as well. As currently implemented in Linux, these operations support batching of contiguous ranges of pages, thereby amortizing the overheads of context switching in and out of the kernel and flushing TLBs. However this is not particularly helpful in our context because the set of pages to be reclaimed simultaneously is rarely contiguous. Thus Eden introduces vectorized variants of `UFFDIO_WRITEPROTECT` and

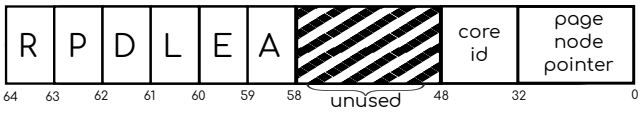

Figure 5: Eden's page metadata format. Each page has six bitflags (Registered, Present, Dirty, Locked, Evicting, Accessed), the ID of the core managing any outstanding faults, and a pointer to the page node for pages present locally.

`madvise(MADV_DONTNEED)`, which enable Eden to reclaim multiple non-contiguous pages in a single syscall and a TLB flush. This significantly improves the scalability of these operations, as shown in Figure 8(a) of our evaluation.

4 Implementation

Our prototype builds on Shenango [45], a user-level thread scheduler written in C/C++. The Eden runtime extends Shenango in several ways. It adds 6,296 lines of code that implement: an `LD_PRELOAD` library to provide transparent support for the standard memory allocation API; fault handling, work stealing and memory reclamation; and page management. Note that while we based Eden on Shenango, Eden can be easily extended to run with other "two-level" thread schedulers like Arachne [39] or GoLang [21]. In addition, Eden includes an RDMA network stack (605 lines) and a far-memory RDMA server (1,304 lines). Finally, we implement a stand-alone tool to identify page fault locations [51] and a patch for the Linux kernel to support vectorized syscalls.

Memory management. Eden intercepts memory allocation functions and forwards them to to `jemalloc` [17], which generates easier-to-manage batched page-sized allocations. Eden supports mapping multiple far memory segments from one or more memory servers as separate memory regions that are registered with `userfaultfd`. Eden maintains a single metadata array per segment with 64-bit per-page entries (shown in Figure 5) that hold page flags, an index to the page node in the reclaim page lists (that is valid only if the page exists locally), and a few bits to save the current application core ID when a fault is in progress. The core ID allows the control core to perform targeted stealing and resolve deadlocks (§3.2). In terms of memory overhead, Eden requires both a 64-bit entry for every mapped page and an additional 24-B page node for resident pages; e.g., supporting 256 GB of far memory with a 64-GB local cache requires 750 MB (∼1%) for Eden's bookkeeping using standard 4-KB pages.

Reclaim policies. Even Eden's basic hints enable more sophisticated reclaim policies than existing kernel-based systems. We implemented four reclaim policies; applications are free to choose the policy that provides the best trade-off for them. With *default*, Eden does not use hints for page hotness and just maintains one big page list that it pops off candidates from. *Second chance* is similar to the Linux kernel's default policy, where hints set a hot bit and Eden maintains two lists and bumps or evicts a page depending on the hot bit. With *LRU*, hints update a timestamp on the page on each access which Eden uses to either evict or bump the page to any of the top $(N-1)$ lists based on how recent the last access was, where N is the configurable number of lists. Finally, with *priority*, Eden lets applications set different priorities for pages faulted in through hints (via the `ev_prio` argument). Eden maintains a configurable number of priority levels for each eviction page list, which are appended to based on the priority set by hints (default is zero for low priority). When evicting, Eden starts with the higher priority levels. This policy generalizes AIFM's non-temporal dereferences, which reclaims an object immediately after its `DerefScope` ends [41].

Fault tracing tool. To find which source code lines should be hinted, we provide a simple tool (also used in [51]) that outputs faulting code locations and the fraction of total page faults they trigger. Like Eden, the tool registers application's memory with `userfaultfd` and records the call stacks on each page fault. While tools like `ftrace` can also be used to collect traces, our tool can run as a part of Eden and help the developer iteratively with the hinting process; when running with Eden, the tool will only record the unhinted faults, and generate flame graphs that highlight only the parts of the application that are yet to be hinted. In future work, we could extend the tool to help detect misplaced hints (e.g., by checking to see if each hint results in page fetches) and automatically suggest hint extensions like `rdahead` and `seq` based on profiled access patterns at the hinted faulting locations. The tool is openly available at `https://github.com/eden-farmem/fltrace`.

5 Evaluation

Our evaluation seeks to address the following questions:

1. How much developer effort does Eden require? (§5.2)
2. How does Eden perform for different applications, and how does it compare to Fastswap, AIFM, and TrackFM? (§5.3)
3. How much do each of Eden's individual elements contribute to its performance? (§5.4)

We answer these questions using four C/C++ far-memory applications: The synthetic Web service and DataFrame library [2] used in the AIFM paper [41], Memcached [32], and an implementation of parallel sort [43]. We use the first two to compare Eden's mechanisms for leveraging application-specific information against those of AIFM (and TrackFM in the case of DataFrame). The latter two represent common real-world workloads. (For brevity, we discuss parallel sort in the Supplementary Material (§A.4), as the results are qualitatively similar to the DataFrame benchmark.)

5.1 Experimental setup

We evaluate Eden and Fastswap on a testbed of three Linux servers, each equipped with a single 100-Gbps Mellanox ConnectX-5 NIC and two 14-core Intel Xeon Gold 5120 CPUs clocked at 2.20 GHz and provisioned with 94 GB of

App	LoC	# Hints	Coverage	Extensions	Reclaim Policy	Max gain%	Benefits from
DataFrame [2]	15k	11	97.3%	`rdahead,seq`	Default	37%	Read ahead
Syn Web Service [41]	1.1k	4	> 99%	`ev_prio`	Priority	178%	Priority reclaim
Memcached [32]	43k	2	> 99%	-	Second-chance	104%	Latency hiding
Parallel Sort [43]	1.2k	6	> 99%	`rdahead,seq`	Default	19.4%	Read ahead

Table 1: For each application, we show the total lines of code (LoC), the number of hints we add (# Hints), the percentage of faults covered by the hints, any hint extensions employed, the reclaim policy (§4) employed, the maximum relative performance gain over Fastswap at any local memory, and the main reason for that speedup.

DRAM per socket. We run on a single NUMA node and disable hyperthreading, TurboBoost, C-states, and CPU frequency scaling. One server plays the role of both the far-memory server and, for certain applications, the client traffic generator. The applications under test run on one of the other two servers: one is configured for Eden and runs Linux 5.15, the other for Fastswap which requires Linux 4.11. We disable page-table isolation (PTI) because it is not available until 4.15, although Eden's performance is similar with PTI enabled.

Both Eden and Fastswap use a dedicated core for memory reclamation, placing them on equal footing. For Eden we use the reclaim policy that worked best for each benchmark (listed in Table 1) while Fastswap uses the Linux default. We employ eviction batching (32 pages) and set the eviction threshold (the local memory usage at which the control core begins to reclaim memory) to 99% of local memory. For Fastswap, the analogous eviction batching parameter is hardwired in the kernel with the reclaim core activating when the memory is above the limit, and other cores joining in if the excess memory use exceeds 2,048 pages.

We run AIFM and TrackFM in the same configurations as their published artifacts on Cloudlab [1, 3]. AIFM requires manual porting and we use only benchmarks ported by the AIFM authors. To facilitate comparisons across all systems, we normalize the results and report the throughput (or runtime) of each system as a fraction of its performance in a fully local baseline configuration, i.e, when no page faults are required. For Eden we measure baseline performance before adding hints (so the normalized numbers incorporate any hinting overhead). In the case of AIFM and TrackFM, we normalize to the performance of versions of the application programs that do not link these libraries to account for their dereference overheads.

5.2 Developer effort

Our analysis in Section §2.3 shows that, for many applications, we can achieve good coverage of potential page faults with a small number of faulting locations. Table 1 summarizes the hinting effort for the applications we evaluate by showing the number of (single-line) hints added, the resulting page fault coverage, and the way hints are extended to benefit each application. We find that we are able to hint faults in Eden for all four of these applications with a handful of hints (at most 11). For DataFrame, the percentage of unhinted fault occurrences is at most 2.7% and for the rest of the applications it remains less than 1%. In the rest of this section, we detail

```
679 new_col.reserve(std::min(sel_indices.size(), vec_size));
680 hint_fault(new_col.data(),new_col.size(),mut=True,rdahead=MAX);
681 for (const auto citer : sel_indices)
682   hint_fault(&(*citer), seq=True, rdahead=MAX);
683   const size_type index =
684       citer >= 0 ? citer : static_cast<IT>(indices_size) + citer;
685   const auto citer = sel_indices[i];
686   const size_type index =
687       citer >= 0 ? citer : static_cast<IT>(indices_size) + citer;
688   if (index < vec_size) {
689       hint_fault(&vec[index], seq=True, rdahead=MAX);
690       new_col.push_back(vec[index]);
```

Listing 1: Lines 680, 682 and 689 are the hints we add to cover the two top faulting locations in DataFrame. Line 680 hints the write faults for the `new_col` vector in 690, whereas 682 and 689 hint the read faults for the lines that immediately follow them. The latter two hints are extended with `rdahead` and `seq` as they access a contiguous vector in a loop.

the hinting process for DataFrame as it is the most complex application we consider and allows direct comparison to the effort required for an AIFM port. The process was much more straightforward for the other applications which we omit due to space constraints. (We summarize the analogous process for Memcached in the Supplementary Material (§A.2)).

DataFrame and AIFM porting. DataFrame [2] provides a Pandas-like C++ library for common data-analysis tasks like slicing, grouping, and aggregating table-based in-memory data structures. Tables are stored in columnar format with STL vectors for each column, and, as such, most data accesses occur through vector indexing. The AIFM authors report changing 1,192 lines of code, or 7.7% of the total codebase to port the library to far memory; most of these changes replace the STL vectors used to implement table columns with custom, far-memory-friendly vectors provided by AIFM's customized STL library. Instead of transforming the data structures, TrackFM [44] (like Eden) interposes on all memory allocations using a pre-defined object size for the entire program. It then uses LLVM-based compiler passes to transforms all pointers, while applying a loop-chunking optimization to reduce dereference overheads which is necessary for this heavy array-scanning application. Because NOELLE [31], TrackFM's LLVM C++ frontend, does not currently support certain C++ semantics in its loop analysis, the authors first port the application to C. (Mira's authors [25] reported a similar effort to simplify DataFrame's C++ constructor and list initializations for which their frontend [33] had limited support.)

Hinting DataFrame for Eden. For Eden, we took the

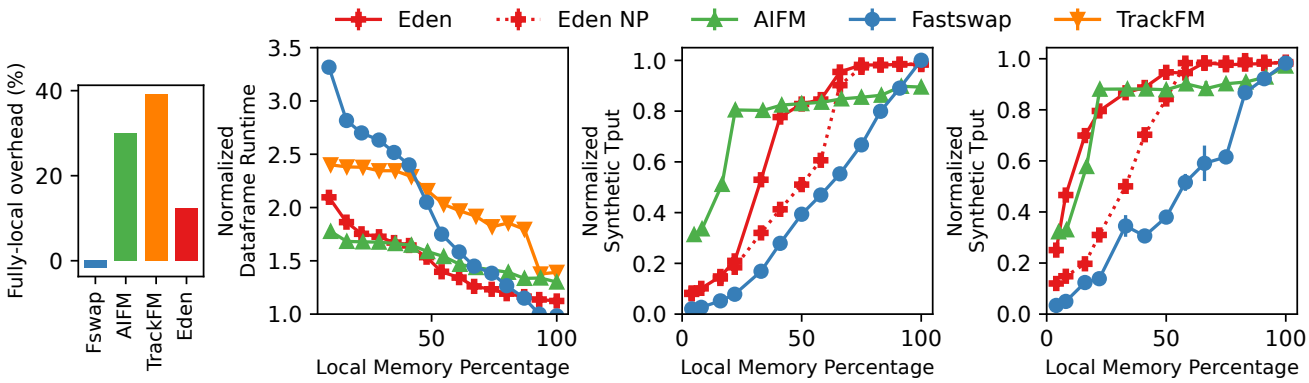

Figure 6: Performance comparison with two AIFM-ported applications. From the left, the first graph shows the performance overhead of DataFrame running on Eden, TrackFM, AIFM, and Fastswap in fully-local configurations compared to its native performance, highlighting the much higher overheads of object-based approaches. The second graph shows the runtime of DataFrame as a function of local memory normalized to the native performance. The third graph similarly shows the normalized throughput of AIFM's synthetic Web server on Eden with and without (Eden NP) priority hints, AIFM non-temporal, and Fastswap. The rightmost graph shows the same experiment with 1,600-byte hash entires.

original, unmodified DataFrame library [2] and traced faults when running with AIFM's analytics benchmark based on NYC Taxi trip data. 23 distinct code locations are responsible for 95% of DataFrame's faults (Figure 2) but we are able to presage two thirds of the faults by adding hints for just the top-three faulting locations. These locations are highlighted by our tool in a list of top faulting locations ordered by fault density. (Figure 10 in the Supplementary Material shows an example fault-density flame graph for DataFrame). Out of the 23 locations, 13 raise allocation faults for newly created column vectors. Because Eden controls memory allocations, we optionally support pre-faulting small allocations, i.e., those less than 100 MB, at allocation time to avoid the need to add explicit hints. With DataFrame, this feature obviates the need to explicitly hint all but one allocation location, leaving only 11 locations to hint.

The top-three faulting locations fall in the selection operator where new columns are generated out of existing ones, and involves a lot of data copying; Listing 1 (without the highlighted green lines) shows the portion of original code where a new column is written to from an existing one (Line 690 sees two faults) based on an index vector (accessed in Line 683 that sees another fault). Unsurprisingly, this is one—and by far the most impactful (c.f. [41, Fig. 8])—of the operations that AIFM offloads to the memory server. We add two hints to read in the index and data vectors, and a third to allocate the column vector in one go.

Extended hints. Each of these hints are inside C++ vector scans, prompting us to add `rdahead` to bring in more pages and `seq` to reduce hinting overhead in each instance. (As we show in Section 5.3.1, this enables Eden to perform on par with AIFM.) Overall, our changes to the application include 11 hints, all similar to those above, each incurring an extra line of source code. With the assistance of our tracing tool, it took the first author only a couple of hours to insert the hints and no time was spent in debugging.

5.3 Performance benefits

We compare the performance of Eden to AIFM, TrackFM, and Fastswap [13], a state-of-the-art kernel-based far-memory system. We evaluate each of our four applications with Fastswap, two with AIFM, and one with TrackFM. Eden consistently outperforms Fastswap, due to the benefits enabled by its hints; for each app, a single mechanism (listed in the last column of Table 1) predominantly drives the performance improvement. Eden's hints can capture most of the benefits of AIFM with many fewer code changes. Specifically, for DataFrame, Eden's hint-level read-ahead was enough to match AIFM's data structure-specific prefetching. In AIFM's Synthetic benchmark, Eden's hint-level eviction priority matched AIFM's data structure-level cache separation. We only evaluate the Synthetic and DataFrame applications on AIFM because the effort of porting other applications to AIFM is high and out of scope given our limited expertise with AIFM. Eden bests TrackFM's DataFrame performance due to the latter's need to insert software guards around every potentially remote access as opposed to Eden's few hint locations.

5.3.1 DataFrame

We start with the performance of the DataFrame application described in §5.2. We use the benchmark provided by the AIFM authors [1], varying the local memory between 3 GB and 31 GB, the workload's full working set size. The leftmost graph of Figure 6 shows the overhead of running DataFrame on Fastswap, AIFM, TrackFM, and Eden in fully local configurations. At 100% local memory, AIFM incurs a slowdown of 30% (85 seconds normalized to the 65-second baseline on AIFM's testbed) because of overheads on pointer dereferences, while TrackFM's overhead is even higher (more so

than reported in their paper [44]) because TrackFM runs a C-ported version of the application while all the other systems run the original C++ version. Eden incurs a 12% (82 seconds normalized to the 73-second baseline on Eden's testbed) overhead due to its localized hints. Fastswap incurs essentially no overhead due to its exclusive reliance on hardware guards.

The second graph of Figure 6 shows the normalized runtime of DataFrame on each system as a function of available local memory.[3] Fastswap's performance degrades significantly due to page-fault overheads whereas Eden, TrackFM, and AIFM degrade more gradually. As local memory becomes extremely constrained, Eden's performance deteriorates slightly due to an increased prevalence of unhinted faults (rising to 2.7% at 9% local memory); yet even then Eden's performance remains close to that of AIFM—achieving a normalized runtime 37% faster than Fastswap—because Eden is able to amortize page-fault costs with targeted read ahead and eviction batching. For example, with only basic hints Eden's performance at 22% (5 GB) local memory is 4× (296 secs) the baseline (not shown), compared to Fastswap's 2.7×. With eviction batching Eden improves to 2.9× (214 secs)—comparable to Fastswap—and adding targeted read ahead drops Eden's runtime to the final 1.75× (127 secs), competitive with AIFM's 1.67×.

5.3.2 Synthetic Web service front end

We demonstrate Eden's ability to perform application-informed reclaim using the Web service described in the AIFM paper [41]. This application services each client request with a series of lookups: it retrieves a 4-byte user ID by doing several lookups (32 in AIFM's benchmark) into a hash table and then an additional lookup retrieves a large (8-KB) object from a user data array. Both objects are compressed with Snappy [20] and encrypted before being sent back to the client. The workload is designed to stress the hash-table data, which is prone to cache evictions by the much larger user data objects, thus hurting performance under oblivious reclaim policies. AIFM's "non-temporal" feature isolates hash-table data from the user data to avoid cache evictions of the former. Eden deliver similar benefits with priority-reclaim hints (§4).

To facilitate comparison to Eden and Fastswap, we "backport" the application to transparent page-based far memory by replacing AIFM's custom far-memory data structures with standard variants, including the "local-only" hopscotch hashtable available in the AIFM repo [1]. Running the backported application through our fault-tracing tool with AIFM's workload identifies two faulting locations in the hash table and one location for the user-data array access. As a baseline (plotted as "Eden NP"), we add 2 basic hints to the Web service, one before accessing the bucket structure in the hopscotch-hash lookup, and a second ahead of accessing the object itself (the key-value objects are stored separately from the buckets). We consider the workload from the AIFM paper: 128 M hash-table entries and 2 M user objects, resulting in a ∼26-GB memory footprint. Client requests are Zipf(0.8) distributed.

The third graph of Figure 6 plots the normalized performance of Eden, Fastswap, and AIFM, each with 10 cores. At 100% local memory, both Eden and Fastswap achieve normalized throughput close to 1.0 (520 KOPS in absolute terms). However, AIFM incurs overhead due to dereferencing its smart pointers; in this workload this occurs much more frequently than hints in Eden, yielding 10.6% lower throughput for AIFM (385 KOPS normalized to 430 KOPS vanilla performance on AIFM testbed) in fully local configuration.[4] At lower local memory where paging is required, Fastswap degrades roughly linearly, down to a normalized throughput of 0.02 (10.7 KOPS) at 4% local memory. Eden's throughput degrades as well, but remains higher than Fastswap (43 KOPS). Unsurprisingly, AIFM degrades more gracefully than either page-based system, maintaining roughly stable performance until about 20% local memory, at which point almost all local memory is required for the hash table.[5]

To mimic AIFM's non-temporal feature, we prioritize reclaim of array data in the array-access hint using Eden's `ev_prio` setting (plotted as "Eden"). Under this configuration, Eden's performance remains competitive with that of AIFM down to about 40% local memory, as the cache miss rate is low (< 1 miss per request for both) and there is no I/O amplification as all the misses are for page-sized array items—hence, the network read I/O (not shown) for both systems is similar at 15 Gbps. Eden also sees the highest normalized performance gain of 0.49 (or 178%) over Fastswap at this setting. As local memory becomes further constrained, both systems begin evicting (4-B) hash-table entries, causing I/O amplification for Eden. Specifically, AIFM is able to reference far memory on a per-object basis, while Eden must pull in—and evict—entire pages at a time, resulting in less efficient local memory utilization. For example, at 8% local memory, AIFM sees an average of 13 misses per request and generates a network I/O rate of 6 Gbps. Eden sees a much higher I/O rate with 1.2 million page faults at 40 Gbps (which saturates our NIC, explaining the performance bottleneck) due to both a higher miss rate (23 per request) and page-sized I/O (i.e., amplification). Recall that each request in this benchmark can see up to 33 far memory lookups.

To isolate this effect, we re-run the experiment with larger hash-table entries. Specifically, instead of a 4-byte entry as before (for which the memory allocator uses a 32-B slab), we use a 1,600-byte entry (requiring a 2-KB slab) while maintaining total hash-table size by decreasing the number of entries to 2 M. In this configuration (shown in the rightmost graph

[3] This is the experiment depicted in Figure 7 of the AIFM paper [41], but we (like [44]) plot normalized runtime because the benchmark does not specify how to compute normalized throughput.

[4] Note that the original AIFM evaluation uses the far-memory-capable variant as a baseline, leading to higher normalized throughput at all local memory percentages presented in their paper [40].

[5] AIFM employs hand-crafted look-aside tables in the smallest (<20%) memory configurations to avoid spilling the hash table to far memory.

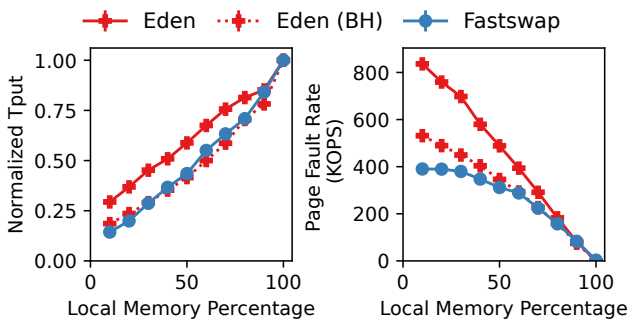

Figure 7: Memcached performance as a function of the fraction of the working set that can fit in local memory. The left-hand graph shows the maximum achievable throughput on both Eden (red; blocking hints in dashed red) and Fastswap (blue) normalized to their respective fully local configurations. The right graph shows the corresponding page fault rate for the same experiment.

of Figure 6), the normalized performance of Eden and AIFM is similar. Hence, we conclude that Eden's poor performance (relative to AIFM) on the original synthetic workload is due to the interaction of Eden's page-based granularity and the workload's lack of (spatial) locality.

5.3.3 Memcached

Next we consider an application that benefits from our user-level scheduler that hides the latencies more effectively than the kernel: memcached. We add only 4 hints using our tracing tool, 3 to cover item search in the hash table in the GET path and an extra write-fault hint in the PUT path. (More details are in the Supplementary Material (§A.2).) To maximize throughput we use a read-only workload and preload memcached with 30M (24-byte, 512-byte) key-value items, resulting in a ∼8-GB memory footprint. We issue client requests based on a Zipf(1.0) distribution.

Figure 7 shows the performance on 5 cores in comparison to Fastswap. We plot the average across multiple runs; the standard deviation is below 2% in all cases, and usually well below 1%. The leftmost graph plots the throughput of each system normalized to the performance of its fully local baseline (4.45 MOPS for Eden and 3.74 for Fastswap). Because memcached requires almost no computation to service each request, its throughput drops dramatically as memory pressure increases. Eden degrades more gracefully than Fastswap, maintaining 29.4% of its baseline performance with ony 10% local memory as compared to 14.4% for Fastswap—an improvement of 104%. (In absolute terms, the gap is even larger: Eden delivers 1.31 vs. Fastswap's 0.54 MOPS.) For this workload, Eden extracts significant additional throughput from the increased concurrency of its lightweight threads (memcached services each request with a separate thread), allowing it to process client requests in parallel on each core, resulting in a dramatically higher page-fault throughput (836 KOPS) than Fastswap (312 KOPS), as shown in the center graph. To isolate this effect from Eden's other features we also measure the performance when Eden spins during fetches ("Eden (BH)") rather than scheduling another thread.

5.4 Microbenchmarks

In this section we isolate the performance benefits of different components of Eden's design through microbenchmarks.

Vectorized operations First we evaluate the performance of Eden's vectorized operations by repeating the experiment from Section §2.2, for the write protect (`UFFDIO_WRITEPROTECT ioctl()`) and unmap (`madvise(MADV_DONTNEED)`) operations. The leftmost portion of Figure 8 shows that with a batch size of 16, Eden's vectorized operations achieve 5.4–6.6× and 3.7–5.7× as much throughput, respectively, as Linux's standard versions. This is because batching enables Eden to amortize the overhead of context switching in and out of the kernel and batch TLB flushes. Eden's vectorized operations are sufficient to sustain line rate of page operations with 100-Gbps NICs, with 1 core for write protecting and 6 for unmapping pages.

Page fetch rate Next we evaluate the maximum rate at which Eden can fetch pages, with eviction disabled. We run a simple multi-threaded benchmark that first registers a unique, large memory region on each core, then touches every page and evicts it to far memory. Next the benchmark accesses the evicted pages sequentially such that each access incurs a major page fault (but local memory is sufficient to never trigger reclaim). We evaluate three systems: Eden, Eden without hints, and Fastswap.

The second graph in Figure 8 shows that, without hints, Eden is bottlenecked by the control core (which must handle every fault), and cannot sustain high throughput. However, with hints, Eden approaches 2 MOPS, achieving 38–88% more throughput than Fastswap, with the same number of cores. This demonstrates the benefit of non-blocking hints, which allow Eden to continue the microbenchmark and initiate additional asynchronous page fetches.

Fetching and evicting Now we adapt the sequential benchmark above by restricting the amount of local memory available so that every page access triggers both a page fetch and an eviction. We evaluate the performance of two reclaim configurations: non-dirty reclaim and dirty reclaim (with write-back) by touching pages with reads and writes respectively, on initialization. Eden uses a batch size of 16. The third graph in Figure 8 shows that throughput with reclaim decreases significantly for both Eden and Fastswap compared to without reclaim (second graph), because reclaim adds more work to every page fault. Though Eden incurs additional kernel crossings during reclaim for write protecting (when the page is dirty) and unmapping, its vectorized APIs enable it to amortize these costs across several pages. In addition, Eden benefits from overlapping the eviction work with page fetches, allowing it to achieve higher throughput than Fastswap.

Read-ahead performance Finally, we evaluate the benefits of leveraging application-specific knowledge with Eden's

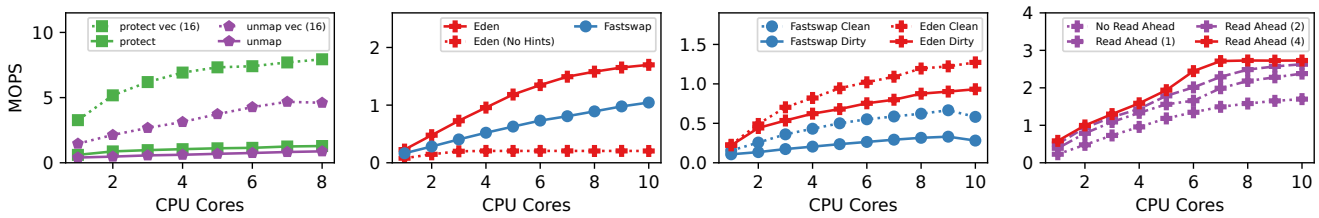

Figure 8: The performance of Eden's vectorized write protect and unmap implementations compared to default Linux (c.f. Figure 1) (a). On a sequential microbenchmark (b,c,d), Eden achieves higher throughput than Fastswap when just fetching pages (b) and both fetching and reclaiming pages (c). In addition, Eden benefits significantly from hints (b) and read ahead (d).

read-ahead feature in its extended hints. We repeat the microbenchmark shown in the third graph ("Eden Clean") with varying amounts of read-ahead, so that Eden can fetch multiple pages in a single RDMA read. The right graph in Figure 8 demonstrates that, with this sequential workload, read ahead improves throughput significantly, saturating the 100-Gbps NIC at high core counts.

6 Related Work

Far memory systems vary widely in terms of programmer effort. Solutions based entirely on transparent hardware guards have been implemented at both the hardware [15, 22, 30] and kernel [9, 10, 12, 13, 18, 23, 28, 38, 42, 49, 50] level, but leave potential application and workload-specific optimizations on the table. In contrast, some app-integrated systems [8, 41, 53] eke out those gains through software guards at the cost of heavy application modifications. Other recent app-integrated systems like TrackFM [44] and Mira [25] use compiler help to automate application changes however they still suffer from significant overhead of their software guards and bring in some practicality concerns like requiring source code for all external dependencies.

Like Eden, DiLOS [52] leverages hardware guards while moving page handling into the application to enable specialized swap caching policies and app-specific optimizations; however, DiLOS employs a LibOS approach to handle hardware guards in the application and cannot run applications natively on Linux. Moreover, while DiLOS's design obviates basic hints, it can still benefit from Eden's extended hints.

Hints, whether specified by the user or a compiler, have long been used to supply prefetching information [34, 37, 46, 48]. Eden's hints go further by both preventing page faults and specifying reclaim behavior. In that way, Eden's hints are semantically analogous to some `madvise()` options. For example, Eden's read-ahead hints are similar to `madvise()` with `MADV_POPULATE` and reclaim priority hints are similar to `MADV_COLD` but with more levels. Critically, Eden's hints are much lower overhead than `madvise()` calls, allowing them to be used more generously in the code. Moreover, `madvise()` guidance is input to the kernel's fixed swap implementation whereas Eden's policies are much more flexible.

7 Conclusion

Eden represents a new point in the design space of far-memory systems: it combines software and hardware guards to intercept accesses to remote memory. Eden uses software guards at the few code locations where most of the remote accesses tend to occur, while relying on hardware guards elsewhere in the application. This approach provides good performance—better than paging-based approaches (pure hardware guards) and comparable and sometimes better than prior app-integrated systems (pure software guards).

It does not, however, tackle a few remaining limitations with page-based accesses. First, for applications that work with small objects and poor locality, managing memory at page granularity can result in overhead due to cache and I/O amplification (as we saw in § 5.3.2). While object-based designs address this issue, such benefits may be easily offset by their high guard and bookkeeping overheads (e.g., object tracking for eviction), which also grow with smaller objects [16]. Second, Eden's far memory handling (miss path) is slower than pure app-integrated systems, especially when Eden triggers a hardware guard and traps into the kernel; in fact, here Eden is even slower than other existing paging-based systems due to its use of `userfaultfd`.

On balance, we believe Eden strikes a useful middle ground between completely transparent page-based systems and existing app-integrated designs. Our approach reduces programmer effort and the overhead of software guards while still enabling application-specific optimizations, which we find more than compensate for the remaining overheads of our page-based design.

Acknowledgements

We thank our shepherd, Sam Kumar, and the anonymous reviewers for their valuable feedback. We also thank Tanvir Ahmed Khan and Baris Kasikci for sharing the dataset [47] we used in our analysis, and Zhenyuan Ruan and Brian Tauro for their help with reproducing the AIFM and TrackFM numbers respectively. This work was conducted under a sponsored research agreement between UC San Diego and VMware Research.

References

[1] AIFM source code and benchmarks. https://github.com/AIFM-sys/AIFM.

[2] C++ DataFrame for statistical, Financial, and ML analysis. https://github.com/hosseinmoein/DataFrame.

[3] TrackFM source code and benchmarks. https://github.com/compiler-disagg/TrackFM.

[4] Linux Userfaultfd. https://www.kernel.org/doc/html/latest/admin-guide/mm/userfaultfd.html, 2022.

[5] Leveldb - an open-source on-disk key-value store. https://en.wikipedia.org/wiki/LevelDB, 2023.

[6] Redis - an open source, in-memory data store. https://redis.io/, 2023.

[7] Rocksdb - a persistent key-value store for fast storage environments. http://rocksdb.org/, 2023.

[8] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote regions: A simple abstraction for remote memory. In *Proceedings of the USENIX Annual Technical Conference*, pages 775–787, July 2018.

[9] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote memory in the age of fast networks. In *Proceedings of the Symposium on Cloud Computing*, pages 121–127, 2017.

[10] Marcos K. Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. Designing Far Memory Data Structures: Think Outside the Box. *Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS 2019*, pages 120–126, 2019.

[11] Masab Ahmad, Farrukh Hijaz, Qingchuan Shi, and Omer Khan. Crono: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 44–55, 2015.

[12] Hassan Al Maruf and Mosharaf Chowdhury. Effectively prefetching remote memory with leap. In *Proceedings of the USENIX Annual Technical Conference*, July 2020.

[13] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Proceedings of the 15th European Conference on Computer Systems, EuroSys 2020*, page 16. ACM, 2020.

[14] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[15] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking software runtimes for disaggregated memory. In *ASPLOS*, pages 79–92, 2021.

[16] Lei Chen, Shi Liu, Chenxi Wang, Haoran Ma, Yifan Qiao, Zhe Wang, Chenggang Wu, Youyou Lu, Xiaobing Feng, Huimin Cui, Shan Lu, and Harry Xu. A tale of two paths: Toward a hybrid data plane for efficient Far-Memory applications. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 77–95, Santa Clara, CA, July 2024. USENIX Association.

[17] Jason Evans. A scalable concurrent malloc(3) implementation for FreeBSD. In *Proceedings of the BSDcan Conference*, April 2006.

[18] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 249–264, Savannah, GA, November 2016. USENIX Association.

[19] Dan Gibson, Hema Hariharan, Eric Lance, Moray McLaren, Behnam Montazeri, Arjun Singh, Stephen Wang, Hassan M. G. Wassel, Zhehua Wu, Sunghwan Yoo, Raghuraman Balasubramanian, Prashant Chandra, Michael Cutforth, Peter Cuy, David Decotigny, Rakesh Gautam, Alex Iriza, Milo M. K. Martin, Rick Roy, Zuowei Shen, Ming Tan, Ye Tang, Monica Wong-Chan, Joe Zbiciak, and Amin Vahdat. Aquila: A unified, low-latency fabric for datacenter networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1249–1266, Renton, WA, April 2022. USENIX Association.

[20] Google. Google's fast compressor/decompressor. https://github.com/google/snappy.

[21] Google. The Go Programming Language. https://golang.org/doc/, 2021.

[22] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. Direct access, High-Performance memory disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 287–294, Carlsbad, CA, July 2022. USENIX Association.

[23] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667, 2017.

[24] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 202–215, New York, NY, USA, 2016. Association for Computing Machinery.

[25] Zhiyuan Guo, Zijian He, and Yiying Zhang. Mira: A program-behavior-guided far memory system. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 692–708, New York, NY, USA, 2023. Association for Computing Machinery.

[26] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 437–450, Denver, CO, June 2016. USENIX Association.

[27] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 514–528, New York, NY, USA, 2020. Association for Computing Machinery.

[28] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. Software-Defined Far Memory in Warehouse-Scale Computers. *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, pages 317–330, 2019.

[29] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. Hpcc: High precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 44–58, New York, NY, USA, 2019. Association for Computing Machinery.

[30] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, page 267–278, New York, NY, USA, 2009. Association for Computing Machinery.

[31] Angelo Matni, Enrico Armenio Deiana, Yian Su, Lukas Gross, Souradip Ghosh, Sotiris Apostolakis, Ziyang Xu, Zujun Tan, Ishita Chaturvedi, Brian Homerding, Tommy McMichen, David I. August, and Simone Campanoni. Noelle offers empowering llvm extensions. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 179–192, 2022.

[32] Memcached. memcached: a Distributed Memory Object Caching System. http://www.memcached.org/.

[33] William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. Polygeist: Raising c to polyhedral mlir. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, PACT '21, New York, NY, USA, 2021. Association for Computing Machinery.

[34] T. C. Mowry, A. K. Demke, and O. Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *OSDI*. USENIX, 1996.

[35] NVIDIA. Mellanox CX-6 100 Gbps NICs. https://www.nvidia.com/content/dam/en-zz/Solutions/networking/ethernet-adapters/connectX-6-dx-datasheet.pdf.

[36] Amy Ousterhout, Joshua Fried, Jonathan Behrens, and Adam Belay. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, pages 361—-377, 2019.

[37] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed caching and prefetching. In *SOSP*. ACM, 1995.

[38] Yifan Qiao, Chenxi Wang, Zhenyuan Ruan, Adam Belay, Qingda Lu, Yiying Zhang, Miryung Kim, and Guoqing Harry Xu. Hermit: Low-Latency, High-Throughput, and transparent remote memory via Feedback-Directed asynchrony. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 181–198, Boston, MA, April 2023. USENIX Association.

[39] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne:{Core-Aware} thread management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 145–160, 2018.

[40] Zhenyuan Ruan. Private communication, 2023.

[41] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-Performance, application-integrated far memory. *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020*, pages 315–332, 2020.

[42] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. Legoos: A disseminated, distributed OS for hardware resource disaggregation. *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018*, pages 69–87, 2007.

[43] Hanmao Shi and Jonathan Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14(4):361–372, 1992.

[44] Brian R. Tauro, Brian Suchy, Simone Campanoni, Peter Dinda, and Kyle C. Hale. Trackfm: Far-out compiler support for a far memory world. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '24, page 401–419, New York, NY, USA, 2024. Association for Computing Machinery.

[45] The Shenango Authors. Shenango's open-source release. https://github.com/shenango/shenango.

[46] A. Tomkins, R. H. Patterson, and G. Gibson. Informed multi-process prefetching and caching. In *Sigmetrics*. ACM, 1997.

[47] Muhammed Ugur, Cheng Jiang, Alex Erf, Tanvir Ahmed Khan, and Baris Kasikci. One profile fits all: Profile-guided linux kernel optimizations for data center applications. *SIGOPS Oper. Syst. Rev.*, 56(1):26–33, June 2023.

[48] S. VanDeBogart, C. Frost, and E. Kohler. Reducing seek overhead with application-directed prefetching. In *ATC*. USENIX, 2009.

[49] Chenxi Wang, Yifan Qiao, Haoran Ma, Shi Liu, Yiying Zhang, Wenguang Chen, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Canvas: Isolated and adaptive swapping for multi-applications on remote memory. In *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation*, April 2023.

[50] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. Tmo: Transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 609–621, New York, NY, USA, 2022. Association for Computing Machinery.

[51] Anil Yelam, Stewart Grant, Enze Liu, Radhika Niranjan Mysore, Marcos K. Aguilera, Amy Ousterhout, and Alex C. Snoeren. Limited Access: The Truth Behind Far Memory. In *Proceedings of the Workshop on Resource Disaggregation and Serverless Computing (WORDS)*, October 2023.

[52] Wonsup Yoon, Jisu Ok, Jinyoung Oh, Sue Moon, and Youngjin Kwon. Dilos: Do not trade compatibility for performance in memory disaggregation. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 266–282, New York, NY, USA, 2023. Association for Computing Machinery.

[53] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. Carbink: Fault-Tolerant far memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 55–71, Carlsbad, CA, July 2022. USENIX Association.

Category	Count	LoC	Max RSS
Parsec [14]	7	10–32K	110–800 MB
Key-value stores [47]	4	7–260K	0.5–1.3 GB
Graph [11]	10	0.2–1.2K	0.1–300 MB
Other	3	1K-15K	>20 GB

Table 2: Application suite for fault location analysis.

A Supplementary Material

We provide additional information about the applications used in our fault analysis (§A.1) and the hinting process for two of our evaluated applications (§A.2 and §A.3).

A.1 Applications

Table 2 lists the number of applications that we evaluate per category and the lines of code and maximum resident set size (RSS) for each.

PARSEC: PARSEC [14] is a benchmarking suite of compute-intensive applications designed to stress parallel performance on multi-core CPUs. We include a subset of these applications which are more HPC focused. We run each with 8 threads and its *native* dataset.

KVS: We evaluate four key-value store applications: two in-memory caches (Redis [6] and Memcached [32]) and two persistent data stores (Rocksdb [7] and LevelDB [5] with memory caches). We use the workloads from LK_PROFILE but increase the number of operations to at least a million, which yields a proportionally larger memory footprint.

Graph: We consider the CRONO graph benchmark [11], which consists of 10 independent multi-threaded graph algorithms. We use the inputs provided with the benchmark suite and run each program with 4 threads.

Other: Contains applications from our evaluation described in Section 5.3. Synthetic and DataFrame are ported from AIFM's evaluation [41], and psort is our own multi-threaded quicksort implementation.

A.2 Memcached hinting

Here we describe the hinting process for Memcached, a widely used in-memory key-value store. Key-value items are sorted into buckets indexed by the hash of the key, and each bucket is a linked list of items storing a key, its associated value, and other metadata like reference counts, locks, and list pointers. To look up a key, Memcached finds the bucket and walks the list until the key is found. We use a port by the authors of Shenango [45] where each request is served by a separate lightweight thread.

We run the executable on our selected workload (see Section 5.3) linked against our tracing tool. The tool produces a flame graph that visualizes the frequency and call stacks for a configurable fraction of faulting locations. Figure 9 shows the locations of 95% of all faults. The bottom-most layer corresponds to `main` and includes all faults, while the top-most layer shows the functions that include the faulting locations.

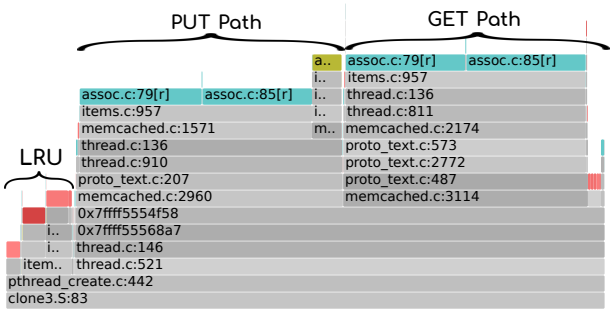

Figure 9: Output of our fault-tracing tool on Memcached.

```
82  item *ret = NULL;
83  int depth = 0;
84  while (it) {
85    hint_fault(&it->nkey);
86    if ((nkey == it->nkey) && (memcmp(key, ITEM_key(it), nkey) ==
          0)) {
87      ret = it;
88      hint_fault(&ITEM\_data(it), it->nbytes);
89      break;
90    }
91    it = it->h_next;
```

Listing 2: In Memcached we add a hint on line 85 to prevent the fault on line 86 and another hint on line 88 to indicate size of the data and ensure that all pages spanned by the item data are faulted in.

Read faults are colored blue, write faults are shown in red, and the protection faults (i.e., the page is present as read only) are greenish yellow. Based on this graph we see that there are two major faulting locations (`assoc.c:79` and `assoc.c:85`) which occur in both the PUT and GET code paths triggered by our 50:50 workload. These correspond to accesses for the bucket and the items. An additional protection fault location appears in the PUT path.

After identifying the faulting locations, Listing 2 shows how we add hints to the item access location, `assoc.c:85`. The fault occurs inside a `while` loop that traverses a linked list in a hash-table bucket: The value of the key being looked up is compared against the contents of the bucket by dereferencing a pointer (`it->nkey`); this dereference frequently causes a fault. We add a basic hint immediately beforehand to prevent this fault. The hint added for the second faulting location (`assoc.c:79`) is similar.

While these two locations cover the majority of faults, there are others. The (write-fault) locations on the far left correspond to accesses by the LRU-maintainer thread. Memcached tracks per-item access information to maintain LRU lists and evicts cold items under memory pressure, thereby dirtying memory even when serving reads. Additionally, it performs "item shuffling" between these lists and updates item pointers in the background, further dirtying memory. As a result, read-only workloads dirty pages at a rate similar to write-heavy workloads. Rather than insert (four additional) hints at these locations, we instead disable both of these unnecessary features in our tests—dramatically improving performance for both Eden and Fastswap—as the underlying far-memory sys-

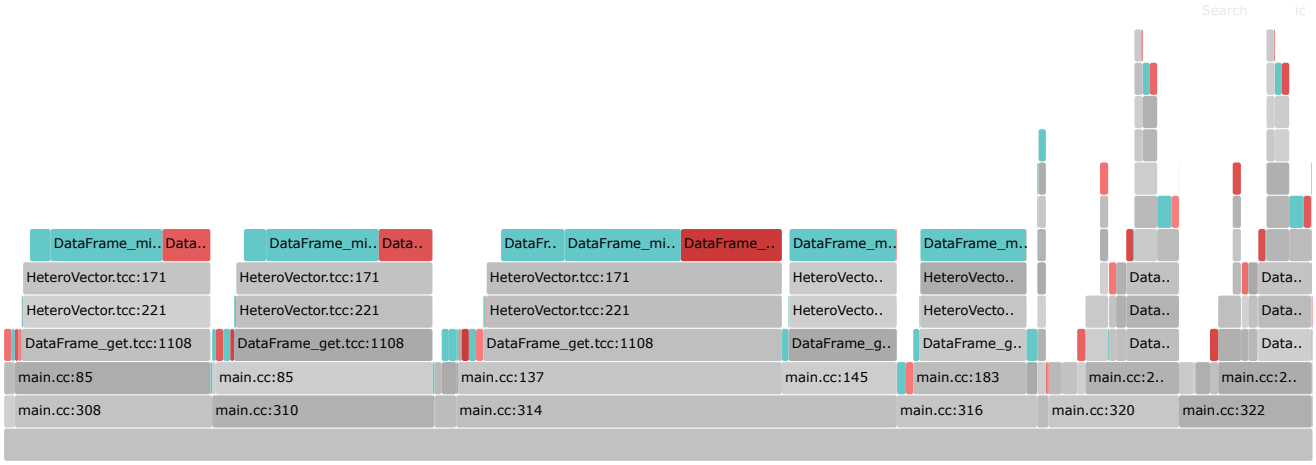

Figure 10: A flame chart for faulting code locations generated from DataFrame's entire execution.

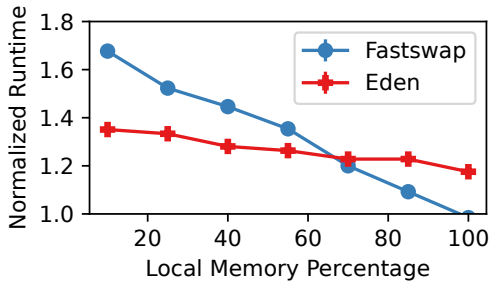

Figure 11: Performance of parallel-sort application with optimal read-ahead settings. The plot shows the run time (lower is better) normalized to the (unhinted) fully local configuration.

tem already takes care of access tracking and moving pages to and from far memory—obviating the need for such internal LRU lists. A number of other rare locations on the right happen when the item found falls across two pages and the page hint brings in only one of them, so we add an extra hint (`assoc.c:88`) for the matched item indicating the size of the item to bring in both pages in such cases.

A.3 DataFrame hinting

Figure 10 presents the fault-density flame graph for DataFrame generated by our tracing tool when running the analytics benchmark. The benchmark runs nine different analytics queries on the data, with seven segments in the flame graph showing the most time-consuming ones.

A.4 Parallel-sort evaluation

Here, we describe the evaluation of a sorting benchmark, in which, unlike the other applications we evaluated, each thread has its own distinct working set. Specifically, we implement parallel sorting with regular sampling [43]. This application first shards an input buffer to individual threads which each quicksort their shard. Then it merges the sorted shards together into an output buffer and copies it back to the input buffer. Because of the overhead of merging results, best performance is typically achieved by matching the number of threads to the number of CPU cores.

Our benchmark sorts 3B 4-byte integers (resulting in a 21-GB memory footprint). We implement two versions: one using Linux pthreads for Fastswap and another using Shenango threads for Eden. We add 6 hints (2 in quicksort, 3 in merge and 1 in copy-back stages) to the baseline code for Eden. The rightmost graph of Figure 11 presents the normalized runtime of both systems running on 10 cores, with Fastswap normalized to the fully local pthreads version and Eden normalized to the (unhinted) Shenango version. (The total runtime does not differ substantively between fully local pthreads and Shenango versions.) Eden experiences a 17.5% slowdown when fully local, most of it coming from the (unnecessary, in that configuration) hints in the partition function of quicksort. However, the (64-page) read ahead signaled through these hints starts helping as memory pressure increases. We plot Fastswap's performance with an optimal *system-wide* configuration of 7-page read ahead. Fastswap extracts less benefit from read ahead because prefetched pages are not mapped until accessed, triggering minor page faults for pages that are already present. As a result, Eden's runtime is 19.4% less than Fastswap's in the 10% local-memory case. Note that the hinting effort, hint extensions and the performance benefits are similar to those of DataFrame as both applications have similar array scanning patterns.