TrackFM: Far-out Compiler Support for a Far Memory World

Brian R. Tauro
btauro@hawk.iit.edu
Illinois Institute of Technology
Chicago, Illinois, USA

Brian Suchy*
brian@briansuchy.com
Northwestern University
Evanston, Illinois, USA

Simone Campanoni
simone.campanoni@northwestern.edu
Northwestern University
Evanston, Illinois, USA

Peter Dinda
pdinda@northwestern.edu
Northwestern University
Evanston, Illinois, USA

Kyle C. Hale
khale@cs.iit.edu
Illinois Institute of Technology
Chicago, Illinois, USA

Abstract

Large memory workloads with favorable locality of reference can benefit by extending the memory hierarchy across machines. Systems that enable such *far memory* configurations can improve application performance and overall memory utilization in a cluster. There are two current alternatives for software-based far memory: *kernel-based* and *library-based*. Kernel-based approaches sacrifice performance to achieve programmer transparency, while library-based approaches sacrifice programmer transparency to achieve performance. We argue for a novel third approach, the *compiler-based* approach, which sacrifices neither performance nor programmer transparency. Modern compiler analysis and transformation techniques, combined with a suitable tightly-coupled runtime system, enable this approach. We describe the design, implementation, and evaluation of TrackFM, a new compiler-based far memory system. Through extensive benchmarking, we demonstrate that TrackFM outperforms kernel-based approaches by up to 2× while retaining their programmer transparency, and that TrackFM can perform similarly to a state-of-the-art library-based system (within 10%). The application is merely recompiled to reap these benefits.

*Now at Google.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 979-8-4007-0372-0/24/04…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Keywords: disaggregated memory, compilers, far memory

ACM Reference Format:
Brian R. Tauro, Brian Suchy, Simone Campanoni, Peter Dinda, and Kyle C. Hale. 2023. TrackFM: Far-out Compiler Support for a Far Memory World. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Applications benefit from deep, hierarchical memories that match the program's available data locality to a memory tier with appropriate performance characteristics. For example, Lagar-Cavilla et al. found that applications access an average of 32% of their pages in Google's warehouse-scale system [20] and most pages are accessed only infrequently. This infrequent access presents an opportunity for a cheaper, slower tier of memory that sits between DRAM and disk. One example of such a *far memory* tier is *remote memory*, alternatively referred to as *disaggregated memory* [1]. In the remote memory model, DRAM on a remote server connected to the local machine with a high-performance interconnect serves as swap space. Remote memory systems accommodate memory-constrained applications by allowing workloads to scale across machines rather than requiring overprovisioning using expensive, large-memory hardware. This reduces ownership costs [40] and mitigates application crashes from unmet memory demands.

Remote memory can be implemented in hardware or software. This paper focuses on software-based remote memory, for which there are two primary techniques: *kernel-based* and *library-based*. The kernel-based approach modifies the OS paging subsystem [3, 13, 44], achieving *programmer transparency*: the application developer gets the advantages of kernel-based approaches for free; even unmodified binaries can benefit from remote memory. Fastswap is a notable example that uses a modified Linux swap subsystem to leverage memory on a remote server using RDMA [3]. The programmer transparency of the kernel-based approach comes at a

cost, however. For example, page fault overheads in the kernel impose a performance penalty on applications relative to using only local memory [35]. The hardware page fault cost creates a fundamental limitation on performance. Additionally, the architected page size of the hardware can poorly match the granularity of application objects: this results in "I/O amplification," where more data is transferred than necessary. Specialized hardware can improve this situation by reducing the granularity of memory faults [7], but this capability is currently limited to research prototypes [32].

The library-based approach to far memory is an important alternative, where developers use modified (or custom) libraries that include data structures designed to leverage remote memory, at granularities appropriate for the application, and entirely in user space. Application-integrated far memory (AIFM) [35] is the exemplar of this approach. AIFM builds on the Shenango runtime's [34] high-performance user-level tasking and networking to hide remote object fetch latencies using prefetching, concurrent fetch requests, caching, and automatic memory evacuation. AIFM can thus achieve considerably higher performance than Fastswap, especially for fine-grained objects. The performance of the library-based approach trades off for programmer transparency, since the application must be reimplemented to leverage remote memory. Implementations like AIFM *do* attempt to insulate application developers to some extent. In the best case, developers need only make minimal changes to their code to leverage remote versions of data structures (e.g., a remote `HashMap`). However, if the AIFM libraries do not provide appropriate data structures, developers must design their own.

The tension between transparency and performance in the kernel-based and library-based approaches creates an opportunity for a third alternative: *compiler-based*. We argue that modern compiler analysis and transformation techniques make it possible to simultaneously achieve programmer transparency and performance. To support this argument, we design, implement, and evaluate TrackFM, a compiler and runtime framework that achieves full transparency using semantics recovered and exploited using state-of-the-art compiler middle-end analyses and transformations, and achieves high performance by using the heavily optimized AIFM runtime as a backend. No specialized hardware or modifications to the OS are required. Using a mix of micro- and macro-benchmarks, we demonstrate that the compiler has sufficient knowledge to allow TrackFM to achieve near performance parity with AIFM (within 10%) while maintaining the programmer transparency of Fastswap.

We summarize our contributions as follows:

- We introduce the compiler-based approach to software-based far memory, which provides a path to simultaneously achieving programmer transparency and performance.
- We demonstrate how to use modern compiler analysis and transformation techniques to automatically transform existing applications to support far memory.
- We introduce new compiler analysis and transformation passes that improve performance for the target applications.
- We present the design and implementation of TrackFM, a new compiler-based far memory system.
- We report on an extensive performance evaluation using numerous microbenchmarks and applications.

TrackFM is freely available online.[1]

```
1 int sum (RemoteArray * array, int n) {
2   int sum = 0;
3   for (int i = 0; i < n; i++) {
4         DerefScope scope;
5         sum += array.at(scope, i);
6   }
7   return sum;
8 }
```

Listing 1. Simple loop using AIFM's remote array.

2 TrackFM Design

Our goal is to use the compiler to approach the performance of library-based far memory solutions by automatically transforming existing applications, eliminating the need for programmer modifications. We aim to reuse the AIFM far memory runtime and automate its integration into the application. As an illustrative example, consider a `for` loop that computes the sum over an array of integers. To make this array remotable in the library-based solution (AIFM), the programmer must use the remote array type provided by AIFM libraries. The programmer must then change their code manually, as shown in Listing 1. The highlighted lines indicate programmer changes. Although these changes are minimal, they require understanding of AIFM's semantics; namely, a scope object must be provided so that AIFM does not evacuate in-use local memory. Moreover, modifying applications with large code bases to run on AIFM may not be practical.

We aim to transform unmodified C/C++ applications to use remote memory *automatically*. Figure 1 shows our overall design. Our compiler toolchain takes the unmodified C/C++ source code[2] for an application, and using an LLVM-based, middle-end analysis and transformation pipeline, remotes certain memory allocations via AIFM. It also injects a thin runtime layer into the application that interfaces with AIFM. The toolchain produces a modified binary that runs on a far memory cluster. Our transformations take place at the IR level.

[1] https://github.com/compiler-disagg/TrackFM

[2] Our approach also applies for applications shipped as LLVM bitcode.

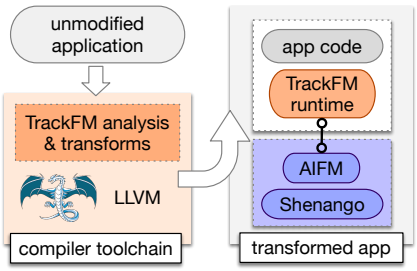


Figure 1. Users compile applications with TrackFM to run on a far memory cluster.

The primary obstacle to automating the integration with AIFM is the semantic gap between the application developer's high-level knowledge of data structures and what the compiler sees at the granularity of memory accesses. AIFM works at the level of *objects*, contiguous chunks of remotable memory, and what constitutes an object is determined by the application developer. For example, when AIFM's object size is set to 256B, a remote 1KB array will be represented by four chunked AIFM objects. A remote linked list, on the other hand, might use an AIFM object size of 64B to constitute a single linked list node. Unlike AIFM, TrackFM works on unmodified code, so it must automatically determine the mapping of memory allocations to AIFM objects using low-level information (i.e., by drawing boundaries around chunks of contiguous memory allocations).

In kernel-based approaches, *any* page can be swapped to a remote node, while in AIFM, candidates for remoting are determined by which data structures the programmer uses the AIFM data types for. Our design strikes a middle ground, where any heap-allocated data can be swapped out (but not at the granularity of pages). Whether these heap-allocated regions actually *are* swapped out depends on temporal access patterns; hot regions will be kept local, while cold ones will be evacuated to the remote node. The TrackFM runtime tracks this "hotness" via AIFM's existing object access interposition mechanisms.

AIFM has several programmer-directed parameters that affect its performance, for example, the degree of concurrency, object size, and prefetching strategy. We will see how the *compiler's* choices for these parameters impact performance in Section 4. Since our compiler framework requires source code, programs that use external libraries present a challenge. The naïve route is to ignore external libraries. Memory that they allocate will not be remotable. However, TrackFM needs to transform pointers to automatically remote them, and those transformed pointers can easily *escape* to library code, which does not know how to handle them. A library may then incorrectly attempt to access remote memory not yet localized by the TrackFM runtime. The alternatives are to (1) have programmers run external libraries through the TrackFM compiler or, (2) only allow pre-transformed versions of the libraries provided by us. In this paper, we explore both options, though the latter is more pragmatic.

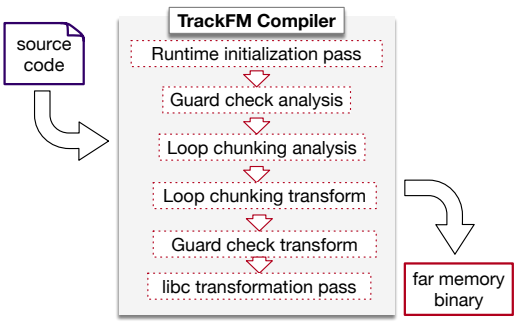


Figure 2. TrackFM's analysis and transformation pipeline.

3 Implementation

We first outline how TrackFM transforms applications to use far memory, then we describe how we incorporate the high-performance AIFM runtime with TrackFM. Finally, we describe our compiler transformations in detail, including how we manage the overheads they introduce. In this paper, we focus on realizing TrackFM in the context of C/C++ programs.

3.1 Far Memory Pointer Transformation

The first distinction that TrackFM must make is between remotable and local-only pointers. AIFM makes this distinction using far memory data structures. However, TrackFM cannot rely on user annotations since we target unmodified code.

Conceptually, all heap-allocated pointers must be managed by TrackFM, and all others (stack, global data, etc.) remain unchanged. However, as a pointer is just an address, we have no *a priori* way to tell them apart. TrackFM does this by overloading the higher-order bits of the address. In particular, it leverages x86 non-canonical addresses.[3] The 60^{th} bit of the address is used to flag a pointer as a TrackFM pointer. If this bit were to be set in any non-TrackFM pointer, the pointer would be invalid. To enforce this distinction, TrackFM provides a custom `malloc` implementation which replaces the default libc `malloc`. Our custom implementation always returns TrackFM (non-canonical) pointers.

Intuitively, a TrackFM pointer can refer to memory that is either on the local or remote system. Thus, the program must be prevented from using the pointer directly. The compiler must provide an indirection layer that, when the pointer is accessed at runtime, localizes the memory and produces a standard pointer in the local address space. Thus, we must

[3]Depending on the x86 implementation, the top 16 or 7 bits of a virtual or physical address must be either all zeros or all ones in order for the address to be "canonical." If a "non-canonical" address is ever used for an instruction fetch, a load, or a store, a general protection fault is triggered.

guard accesses to TrackFM pointers. These guards constitute compiler-injected code that ensures memory is localized before access; they comprise the lion's share of TrackFM's overheads, as we will see in Section 4. To properly guard pointers, the TrackFM compiler applies a series of analyses and transformations at the compiler's IR level (called passes), as shown in Figure 2. These passes are built on NOELLE [27], a novel analysis and transformation framework that expands LLVM [21] by introducing high-level and program-wide abstractions. We discuss each pass below.

Runtime Initialization. To make far memory transparent to programmers, this pass inserts hooks in the program's `main` function to initialize TrackFM's runtime system.

Pointer guards. In this pass, TrackFM searches for all LLVM IR-level load and store instructions that correspond to heap allocations (returned by `malloc`) and marks these instructions as eligible for guard transformation. The pass ignores accesses to stack and global objects by leveraging NOELLE's program dependence graph abstraction, which is powered by several high-accuracy memory alias analyses. Candidate heap pointers are later transformed by the guard transformation pass, described in Section 3.3.

Loop Chunking. We introduce a novel loop chunking analysis to reduce guard overheads introduced in loop bodies. Our loop chunking pass incorporates NOELLE's profiling facilities when available to further improve our optimization. We describe the relevant transformation in Section 3.4, and techniques to improve it in Section 4.2.

Libc Transformation. This pass transforms all memory allocation calls (mainly for heap allocation) in libc (e.g., `malloc`, `realloc`, `free`), into TrackFM-managed memory runtime calls. The TrackFM versions leverage AIFM's region-based allocator under the covers to allocate remotable memory. Custom heap allocators are not currently supported, but provided they simply replace libc malloc with their own managed heap, this would be trivial to add support for. We consider more complicated heap setups involving `mmap()` (e.g., using `MAP_SHARED`) out of scope for this paper.

3.2 Bridging AIFM with the Compiler

To integrate with AIFM, we use a lightly modified version of AIFM that includes hooks into the TrackFM runtime. We next discuss details about integrating TrackFM pointers with the AIFM runtime. In particular, we must transform contiguous heap allocations into AIFM *objects*, fixed-size chunks that can be either in the local or remote state. We will see in Section 3.3 that significant complexity arises because a given heap allocation can comprise multiple AIFM objects, each of which may be in different states (local or remote).

AIFM manages remotable memory at the level of individual data structures. Each of these data structures in the AIFM runtime is implemented as a C++ class which extends a base class that handles the underlying mechanisms of remote objects. We extend this base class with a unified *abstract* data structure (ADS) that the compiler uses to capture all remotable allocations for the application. With AIFM, programmers specify remote memory usage by leveraging one of these specialized data structures. However, with TrackFM, the compiler identifies *all* remotable allocations and attaches them to a single runtime-managed object pool. The ADS thus contains a pool of objects that represent the total far memory that an application can use. TrackFM interposes on an application's allocation sites and chunks the allocations into objects in the global pool at run-time.

Object size selection. In AIFM, the user/data structure developer annotates each data structure with an object size for a given application. Since TrackFM does not require programmer changes, it is currently constrained to choose a single object size at compile time for the entire application. Unlike Fastswap, which is constrained by the page size, TrackFM supports object sizes smaller than a page, mitigating I/O amplification. While multiple object sizes are possible, this increases the complexity of the runtime system and compiler transformations, so we leave this for future work. We note that it is likely the case that only a few fixed object sizes make sense, and that these are likely to be powers of two ranging from 64B (cache line size) to 4KB (base page size). Using object sizes smaller than a cache line would saturate the network with many small packets, and would not take advantage of the network's bandwidth, which is geared to larger packets. On the other hand, much larger object sizes would suffer from I/O amplification, and defeat the purpose of sub-page granularity far memory. While the choice of object size is currently selected by us, the small search space suggests that an autotuning approach is feasible. Furthermore, if we are correct that only the powers of two from 6 (cache line) to 12 (base page size) need to be considered, an exhaustive search involving recompilation and a short-term execution would simply expand the short compile times.

Allocating far memory. TrackFM only remotes heap allocations and maintains a simple non-canonical address space to service memory allocation calls by the application. All memory allocation call sites within libc are intercepted by TrackFM and will return TrackFM-managed pointers starting from the non-canonical address range (starting at address 2^{60}). Because TrackFM rewrites pointers at the middle-end, even if a pointer is cast to an integer type (for example to perform offset math), the resulting load/store will still be properly guarded, provided that the non-canonical bits of the address are preserved. Internally, TrackFM maps non-canonical pointers to objects in an ADS. The object corresponding to a TrackFM pointer can be derived by dividing the TrackFM pointer by the object size (a right shift for powers of two). A single memory allocation can span multiple

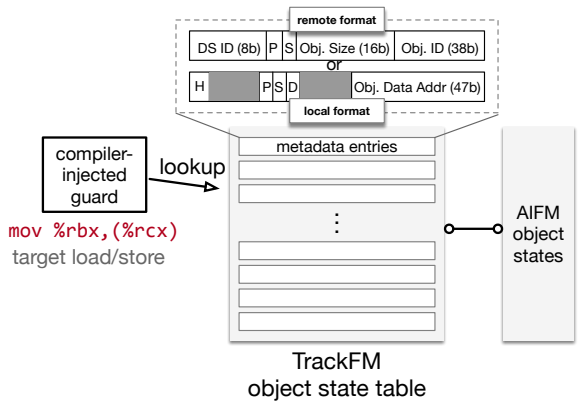


Figure 3. The object state table caches AIFM object metadata (shown on top, and reproduced from the AIFM paper [35]) for lighter-weight guards.

objects, while smaller allocations are grouped into a single object.

TrackFM object state table. Any particular allocation could be in a superposition, i.e., some of its constituent objects (chunks) could be local while others are remote.[4] AIFM tracks the local/remote state of objects by maintaining two metadata representations (one for each state) internally. Determining this state in AIFM requires two memory references, one to find the object, and another to access its metadata. TrackFM eliminates one of these operations by maintaining an *object state table*, an optimization that caches object metadata in a contiguous lookup table, allowing us to perform a simple index calculation rather than an indirect memory reference to derive object metadata. This is possible because of the way TrackFM encodes object IDs in the non-canonical range of the pointer. We modified AIFM so that this table is kept coherent with the AIFM-managed object metadata. The object state table contains metadata entries (8B each) for each object in the system, where the total number of objects is determined by the total size of the remote heap. The overhead of the table can be computed similarly to a single-level page table. For example, if we have a 32 GB remote heap (as in many of our experiments), we would need 2^{23} entries in the table (assuming each object is 4KB), thus consuming 64 MB for the full table. As shown in Figure 3, the compiler-inserted guard derives the object metadata from this table in order to determine whether or not the referenced object is localized.

3.3 TrackFM Guards

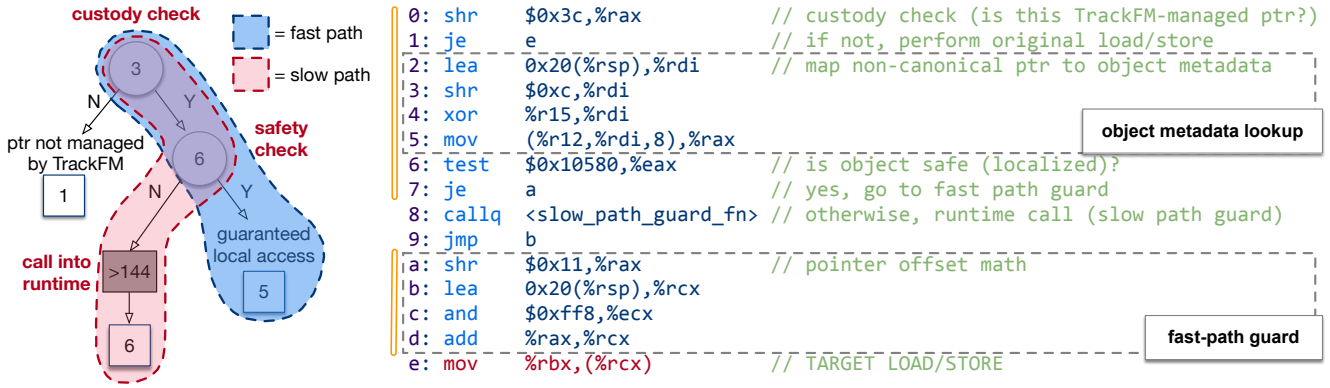
As described above, TrackFM instruments application derived LLVM bitcode with *guards* on every relevant load and store instruction referring to heap-allocated memory at the LLVM middle-end layer. These guards comprise compiler-injected instructions that ensure the memory is *localized* (brought into local memory) before being accessed. TrackFM guards localize an object by reverting the non-canonical address returned from the TrackFM allocator back into a canonical address before execution of the target load/store. Figure 4 depicts the guard. Figure 4a shows an abstract depiction of the injected code as a control flow graph, and Figure 4b shows the guard after it has been lowered to x86_64 assembly. We break down the TrackFM guard into three components: a *custody check*, a *fast-path guard*, and a *slow-path guard*. Note that on the fast path only one of those instructions is a data access (to the object state table) that can result in a cache miss. Figure 4b highlights the fast path through the guard with vertical orange lines on the left. Note that we can also enable optional debug instrumentation that indicates when guards take the fast or slow path, and which AIFM code path they trigger.

Custody check. TrackFM first checks whether the pointer is managed by TrackFM. Recall that this means only heap-allocated memory. If a pointer is *not* managed by TrackFM, we immediately jump to the target load/store. This path constitutes roughly four instructions. If the pointer passes the custody check (i.e., it is a TrackFM pointer), we perform a table lookup to derive the object state table entry corresponding to the AIFM object, and then load the object state of the TrackFM pointer. This path constitutes roughly six instructions.

Fast-path guard. We use AIFM's internal object metadata to determine if an object is *safe* to access, i.e., guaranteed to be local. Safety is satisfied if certain bits in AIFM's internal metadata representation are cleared.[5] When safety is satisfied, the fast-path guard will be taken, constituting 14 instructions. Note that it appears that there is a time-of-check to time-of-use issue between the `test` instruction (line 6) and the actual target load/store (line e). That is, if the safety check passes and this application thread gets context switched out (or even if there is a race), an evacuator might run on another core and delocalize the object, rendering the pointer invalid for the final target load/store. This issue is prevented because AIFM's evacuator threads use a barrier that waits on all application threads to converge to a state where remotable pointers are "out-of-scope." While within the context of a TrackFM guard, the app thread is guaranteed not to be in this "out-of-scope" state, preventing the convergence necessary for the evacuator to proceed. This means that between line 5 and line e, the object cannot be evacuated.

[4] This is a property that makes compiler-based far memory different from prior DSM-focused systems.

[5] AIFM must indicate that the object has been localized either through a blocking access or an asynchronous prefetch request, and is not a candidate for evacuation to the remote node.



(a) abstract control flow. **(b)** generated x64 code.

Figure 4. Left: control flow of compiler-inserted guard check. Circles indicate conditional branches and squares indicate exit nodes. Each node is annotated with the number of x64 instructions executed. **Right**: guard lowered to x64 code. The vertical orange lines indicate the fast path (highlighted in blue on the left).

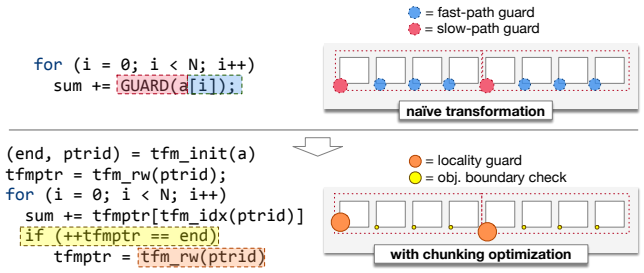


Figure 5. The loop chunking optimization eliminates fast-path guards within loops when object boundaries are not crossed. This trades off a cheaper conditional branch inserted in every iteration (yellow) and a more expensive guard at object boundaries (orange).

Slow-path guard. If the object is *unsafe* to access, then we must call into the TrackFM runtime. TrackFM in turn calls into the AIFM runtime to dereference the object, which could involve a remote fetch. When TrackFM interfaces with AIFM here, it adheres to AIFM's internal `DerefScope` API (shown in Listing 1), and also triggers a periodic collection point to allow stale objects to be evacuated to the remote node. This runtime call in the slow path, which has a higher cost, ensures safety.

Once TrackFM ensures safety, it performs the target load/store. The slow-path guard comprises at least 144 instructions when the pointer object is already localized. However, if the object is remote, the cost of the slow-path guard will be dwarfed by the remote fetch cost.

3.4 Managing Loop Overheads

Up to this point, we focused on direct pointer accesses. However, there are many cases where pointers are accessed via an offset, a major example being array accesses. It is common for such accesses to occur in loops. Ideally, when iterating over a collection (e.g., an array) in a loop, we could localize the entire array at the beginning of the loop, bringing any remote elements local before accessing them. This optimization was commonly employed by compiler-based DSM frameworks [26, 28, 31]. However, because we build on AIFM, and a single collection might constitute *multiple* AIFM objects, the entire collection might be in a superposition (simultaneously local and remote). Moreover, the entire array may not fit in memory. This renders the DSM-style hoisting optimizations ineffective, and it means that all pointer accesses *within a loop body* must be guarded.

However, when many collection (array) elements fit within a single AIFM object, many of these guards are redundant. They are only necessary when we cross object boundaries in the loop. In AIFM, the iterator classes developed by the library developer for the remote data structures manage this overhead. With TrackFM we leverage the compiler's knowledge of the loop to reduce this overhead by developing a *loop chunking optimization* for TrackFM pointers.

Figure 5 depicts such a situation with a contiguous array, where multiple array elements fit within an AIFM object. The naïve guard insertion strategy will involve injecting guards at every element access. The slow-path guards (shown in red) will be taken at object boundaries, i.e., when `i` is a multiple of the object size, and fast-path guards (blue) will be taken on every other access. With our optimization, the compiler can determine the induction variable of a loop, including the step count and the start value of the induction variable, so it knows that sequential accesses within the boundaries of an

already fetched object do not require fast-path guards. This trades off many fast-path guards with a slightly more expensive *locality invariant guard* at object boundaries that calls into the runtime to pin the object in local memory for the duration of accesses to the object (one loop chunk). *Object boundary checks* (yellow) are also inserted on every access to detect when the locality invariant guard should be taken. Note that this optimization is not just applicable to contiguous arrays; it applies more generally to loops that employ a loop-governing induction variable, which is common in practice (we will see this in Section 4.5).

The analysis pass for the loop chunking optimization searches for spatially local memory accesses that occur in loops, typically a popular location of hot code. Upon finding these accesses, TrackFM attempts to mitigate the overhead from guards in the loop body by chunking the original pointer into object-size chunks. To identify such memory accesses, TrackFM makes use of NOELLE's induction variable (IV) analysis.[6] Such analysis is unique as it detects induction variables as patterns in the dependence graph, rather than building on variable analysis. This leads us to capture significantly (~ 3×) more induction variables than what is traditionally possible. However, TrackFM can also be adapted to use other IV analyses should better techniques arise. Note that there is not a correctness issue if the IV analysis misses induction variables; it just results in lost loop chunking optimizations. We plan to further generalize our loop analysis in the future, for example by adapting polyhedral methods [43] to NOELLE.

Our optimization is particularly effective for workloads that display high regularity.[7] Prefetching plays an important role in such workloads. TrackFM can detect sequential access at compile-time, so it uses prefetching alongside loop chunking to mitigate loop overheads. This has an increasing impact on performance as the number of pointers iterating over induction variables in a loop increases. This demonstrates a strength of the compiler-based approach to far memory: kernel-based approaches cannot take advantage of such loop-centric memory analysis; they must make *post hoc* inferences based on run-time page faults.

Improving Loop Chunking. Loop chunking is not *always* beneficial. In particular, when array elements are large (so that fewer of them fit within a fixed-sized AIFM object), or the loops have a small iteration space, there are fewer fast-path guards in the first place. If we apply the loop chunking transformation in such cases, performance can actually drop relative to the standard guards. Intuitively, there is a break-even point when sequential array access occurs at a fine enough granularity for this transformation to pay off. To help the compiler determine where that point is, we develop a simple cost model.

Cost Model. Let o be the size in bytes of a TrackFM object, and let e be the size of an element in a collection accessed in a loop. For example, for an 8-byte integer, e would be 8. We model the number of elements that fit within a single TrackFM object as the *object density*, $d = \frac{o}{e}$. We are interested in determining how densely elements must be packed before the compiler applies the loop chunking transformation. Intuitively, the more dense an object, the more fast path guards will be involved, so the more advantageous the optimization will be. Conversely, if there are few elements per object, the transformation could be detrimental. With the naïve transformation, each loop will iterate over some number of objects, and each object must incur a fast-path guard for each element access, except for the first, which requires a slow-path guard. For each object there will thus be one slow-path guard and $d - 1$ fast-path guards. Slow-path guards have cost c_s and fast-path guards have cost c_f. We model the guard costs at the level of individual objects. We can then estimate the cost of the entire loop in terms of guards:

$$C = (d-1)c_f + cs \tag{1}$$

Our loop chunking optimization replaces fast path guards (14 instructions) with less expensive *object boundary checks* (3 instructions) that determine when an object boundary is crossed. The object boundary checks are shown as small, yellow circles in Figure 5. Slow-path guards are replaced with slightly more expensive *locality invariant guards* (orange circles) at object crossing boundaries, which involve a call to the runtime. We model the cost of the boundary checks as c_b and the locality invariant guards as c_l. The cost of the transformed loop in terms of guards is then:

$$C_{opt} = (d-1)c_b + c_l \tag{2}$$

When a loop iterates over large elements, the relatively high cost of the invariant guard can offset the elimination of the fast-path guards. Thus, we must only apply the optimization when there is sufficient object density, i.e.:

$$d > \frac{c_s - c_l}{c_b - c_f} \tag{3}$$

Figure 6 shows the projected cost of a simple loop with a varying number of iterations for the baseline method and the loop chunking optimization. The chunking optimization becomes preferable once an object comprises as few as ~730 elements. The curve on the plot shows empirical measurements of loop cost. Note that the projected break-even point matches the empirical data. Thus, if the compiler can determine d, we can make intelligent choices about when to apply the loop chunking transformation. To do this, we leverage

[6] This is a more sophisticated analysis than what is available in gcc or LLVM. See §2.B and §4.C of the NOELLE paper [27] for details.

[7] That is, spatial locality of access closely matches temporal locality of access.

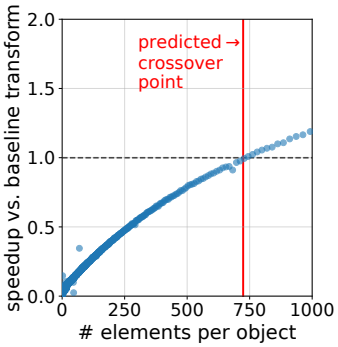


Figure 6. Cost model to capture the point at which loop chunking becomes advantageous. The horizontal dotted line shows empirically when loop chunking benefits, and the vertical red line shows when the model predicts a beneficial outcome.

NOELLE's profiling engine to collect loop code coverage statistics. With the profiling pass in TrackFM we filter out loops with low object density transparently without modifications to source code.

4 Evaluation

The TrackFM compiler must make choices about how it structures far memory objects and passes information to the runtime system. We evaluate the performance impact of these choices and the overheads of compiler-injected guards using microbenchmarks, studying the impact of different types of workloads and access patterns in a controlled setting. We then demonstrate that by making good choices, TrackFM can approach the performance of AIFM on application benchmarks while maintaining programmer transparency. We seek to answer the following questions in our evaluation:

- How expensive are TrackFM guards? (§4.1)
- To what degree can compiler analysis and transformations mitigate guard overheads? (§4.2)
- When the compiler can control AIFM object size and prefetching, how do its choices impact performance? (§4.3)
- To what degree can TrackFM mitigate I/O amplification? (§4.4)
- How do TrackFM's optimizations translate to overall application performance relative to state-of-the-art approaches? (§4.5)
- How does TrackFM affect code size and compilation time? (§4.6)

Experimental setup. We conducted our experiments on CloudLab [10] using two x170 machines with 10-core Intel Xeon E5-2640v4 CPUs clocked at 2.40 GHz, 64GB RAM and a 25 Gb/s Mellanox ConnectX-4 NIC. We used Ubuntu 18.04 (to support AIFM) with stock Linux kernel version 5.0 and DPDK version 18.11. For Fastswap measurements we use the latest version[8] ported to the 5.0 kernel.[9] We use the most recent publicly available version of AIFM.[10] TrackFM builds on LLVM version 9.0.0, with NOELLE v9.8.0. For C++ applications, we use libc++ version 9 provided by clang (we directly compile it with TrackFM). For large codebases we use WLLVM[11] to produce bitcode for the entire application before passing it to the TrackFM compiler.

TrackFM Guard Type	Cached	Uncached
TrackFM fast-path read guard	21	297
TrackFM fast-path write guard	21	309
TrackFM slow-path read guard	144	453
TrackFM slow-path write guard	159	432

Table 1. TrackFM fast-path vs. slow-path guard costs when a object is local. Costs are reported in median cycles over 1000 trials.

Runtime Event	Local Cost	Remote Cost
Fastswap read fault	1.3K	34K
Fastswap write fault	1.3K	35K
TrackFM slow-path read guard	453	35K
TrackFM slow-path write guard	432	35K

Table 2. Comparison of primitive overheads for TrackFM and Fastswap. Costs are reported in median cycles over 1000 trials.

4.1 Guard Overheads

The primary source of TrackFM's overhead comes from the compiler-inserted guard instructions at the bitcode level on heap-allocated loads and stores. Table 1 shows their costs in cycles relative to local load/store operations. The additional overhead for a fast path guard relative to a local unmodified load/store (36 cycles) instruction is 21 cycles. This will be the common case for applications that have locality of access. The uncached slow-path and fast-path guards are more expensive, but better than a page fault.

The slow-path guard is similar in cost to a major page fault in Fastswap when an object is not present in local memory because both events trigger a remote fetch over the network. For reference, Table 2 compares slow-path guards to remote page fault costs in Fastswap (both when the page is local and remote). Handling a page fault in the kernel incurs 2.9× the cost of handling a slow-path guard in TrackFM when the data is local. This changes when the object/page is remote due to Fastswap's fast RDMA backend, which outperforms our use of AIFM's TCP-based backend (from Shenango) when

[8] commit 9cfc2a
[9] https://github.com/nilyibo/fastswap, commit 9d5c6f
[10] https://github.com/AIFM-sys/AIFM, commit aaf711
[11] https://github.com/travitch/whole-program-llvm

there is not sufficient concurrency. However, even with this high-performance networking layer, Fastswap still provides little benefit over our remote slow-path guard. This is due to Fastswap's page fault handling overheads (e.g., mapping and cgroups memory reclamation).

If we really instrument every load and store to heap-allocated memory, what would the costs be? To provide initial intuition, we used TrackFM to automatically transform the STREAM benchmark [29], which has a 9GB working set. This transformation produces up to 56 million slow-path guards and ~10 *billion* fast-path guards. Note that we must pay the cost of these guards *even when objects are local*. Neither kernel-based approaches nor library-based approaches pay such costs for local objects (though AIFM does incur overhead for smart pointer indirection). Thus, it would seem that these guards present an insurmountable barrier to achieving good performance. However, as we will see in the next section, TrackFM can exploit regularity in the workload to dramatically reduce the number of guards.

4.2 Mitigating Guard Costs

To make compiler-assisted far memory feasible, there are two paths to increase performance: (1) reduce guard costs and, (2) reduce the *number* of guards. We spent significant effort on (1), making the common case fast-path guard involve a small number of instructions (only 14). In this section, we focus our discussion on the second path.

Loop chunking transformation. Loop chunking, described in Section 2, eliminates fast-path guards, a key factor for improving performance. To understand its impact, we first evaluate its effects on the STREAM benchmark, which involves sequential access to arrays of small elements (integers), and is simple to transform. The "Sum" test consists of a single memory access to an array element (`sum+=a2[i]`) within a loop. "Copy" consists of two memory accesses (`a1[i]=a2[i]`) within the loop body. Figure 7 shows the speedup when using the optimized loop chunking transformation relative to the naïve transformation, where every loop element involves a fast-path guard. The total working set size for both examples is fixed at 12GB to aid in comparison. Note that the local memory constraint enforced on the application does not include the metadata used by AIFM/TrackFM.

We see that as the number of memory accesses within the loop increases (looking at the figures top to bottom), the speedup offered by loop chunking increases due to the larger number of fast-path guards that are eliminated. For example, for "Sum," we reduce the fast-path guard count from ~1.6 billion to *zero*. Notice that the horizontal axis sweeps the amount of local memory available to the application, with increasing memory pressure to the left. These graphs tend to have an inclination towards the right-hand side since in that regime the system is less network-bound, so the importance of eliminating guard overheads is amplified.

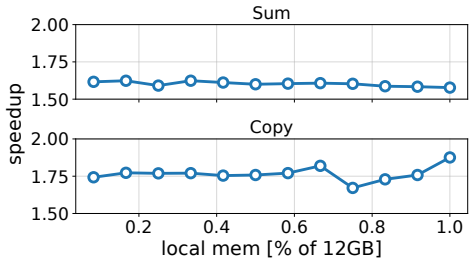


Figure 7. Speedup from loop chunking improves with increased memory accesses in loops as more fast-path guards are eliminated.

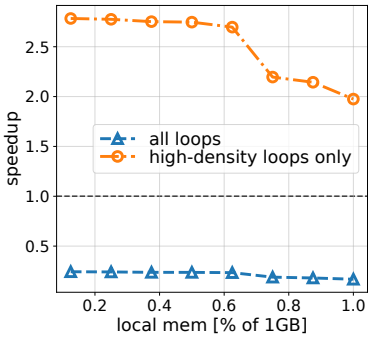


Figure 8. TrackFM can selectively apply the loop chunking optimization (like in *k-means*) to avoid collections with low object density.

Improved Loop Chunking. To showcase how profiling can be coupled with our cost model from Section 3.4, we automatically transformed a *k-means* benchmark, which contains many loops for which it would be detrimental to apply the loop chunking transformation. We run *k-means* with 30 million points. The working set size is fixed at 1GB.

Figure 8 shows the results of applying the loop chunking optimization indiscriminately to *all* loops compared to applying it only to those loops identified as viable candidates by the TrackFM profiler, according to our cost model.

Both lines are normalized to the baseline (no loop chunking) to measure speedup. The figure shows that applying the loop chunking transformation indiscriminately produces poor results and suffers on average 4× slowdown. This is because *k-means* has many nested loops with a low object density. Such nested loops amplify the cost of loop chunking. In this case, there were at least 512 array elements per AIFM object. The chunking optimization detects 103 array pointers, and after applying the cost model only 27 were optimized. Applying the cost model to the loop chunking

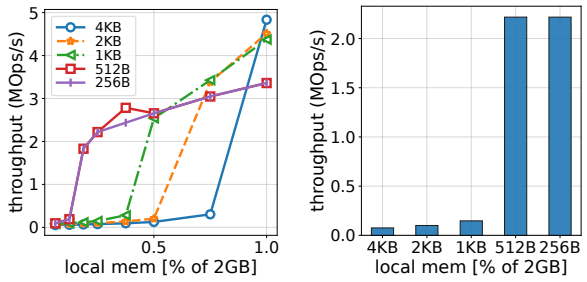


Figure 9. Impact of object size on STL maps. Fine grained memory accesses with little spatial locality can benefit from small object sizes.

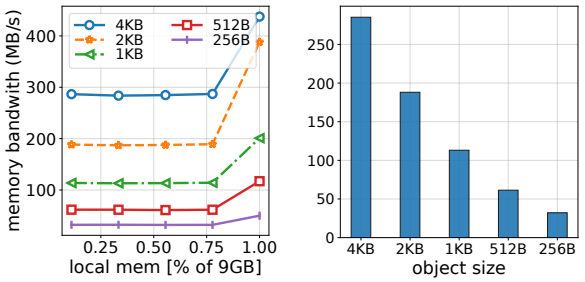


Figure 10. Impact of object size on STREAM. Access patterns with high spatial locality benefit from the choice of a larger object size.

pass here improves the situation considerably, resulting in a mean speedup of 2.5×.

4.3 AIFM Parameters

The TrackFM compiler must make two primary choices when integrating with AIFM: the object size and prefetching strategy. This section explores the impact of those choices.

Object size. TrackFM currently chooses an object size at compile time, though this choice could in principle be informed by profiling. To evaluate the impact of this choice, we compare two microbenchmarks with different degrees of spatial locality and granularity of access.

The first microbenchmark involves accessing a hashmap, much like how a key-value store would operate. We use the unordered hashmap implementation from the C++ STL. Both keys and values are 4B integers. In this case, the entire C++ STL is transformed by the TrackFM compiler. The working set size is 2GB. We use a workload generator to access the hashmap (50 million lookups) according to a Zipfian distribution with skew 1.02. To generate the access trace, we store a sequence of keys sampled from the distribution in a separate 190 MB array also allocated on the heap.

In this case, a small handful of the entries in the hashmap will constitute the majority of accesses, so there will be a high degree of temporal locality (but little spatial locality), and accesses occur at very small granularities (4B). The left side (Figure 9a) shows the impact of varying object size as we sweep the amount of local memory available, and the right side (Figure 9b) highlights the impact for a fixed proportion of the working set size available to local memory (25%). We measure the throughput (MOps/s) of the generated workload. In this case, a smaller object size is clearly preferable.

If we look again at STREAM, where the access pattern shows almost perfect spatial locality, we would expect to see different results. Here, we use the "copy" benchmark from STREAM with a working set size of 9GB. In this case, we measure the far memory bandwidth (the default metric

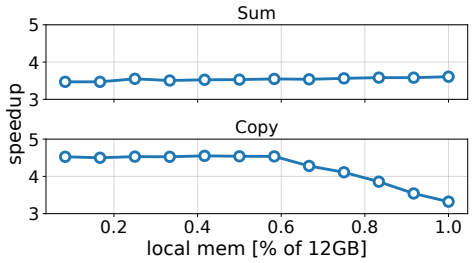


Figure 11. Speedup of prefetching coupled with loop chunking vs. only loop chunking. The combinations helps TrackFM extract more performance from workloads with spatial locality.

reported by STREAM). Though the granularity of access for this example is even smaller (integers), the high degree of spatial locality necessitates chunking elements into larger objects. In this case, 4KB is the better choice.

Figures 9 and 10 highlight that proper selection of object size is critical to performance. While we currently make this choice offline, we envision using profiling to make this choice when application code is recompiled with TrackFM.

Prefetching. When much of the application's memory is remote, the costs of remote fetches can dominate execution time. To mitigate network costs in this regime, TrackFM must employ prefetching to exploit spatial locality. We again run an experiment on STREAM, this time with and without prefetching enabled. In this case, we use AIFM's existing stride prefetcher, and we prefetch pointers operating on induction variables as identified by TrackFM's loop chunking pass. Figure 11 shows the speedup of using prefetching relative to no prefetching as we sweep the amount of local memory available. The loop chunking optimization discussed previously is enabled in both cases. If we focus on the left-hand side of the figures (where remote costs dominate), we see a large impact (almost 5×) on overall performance. As

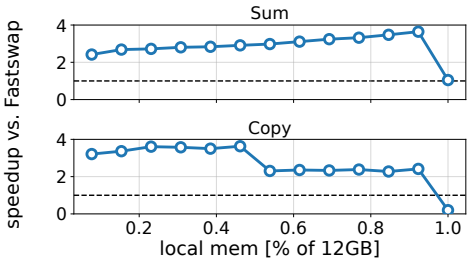


Figure 12. Speedup on STREAM relative to Fastswap with prefetching and loop chunking enabled. TrackFM's memory analysis helps to best exploit AIFM's high-performance prefetching.

more local memory is available, the cost of guards dominate, so the impact of prefetching reduces. We validated this with experiments (not shown for space) that demonstrate that the relative number of *critical* remote fetches, i.e., the number of loads/stores blocked by first having to fetch the object from remote memory when prefetching is disabled, is reduced dramatically with prefetching.

Figure 12 shows the speedup relative to Fastswap on STREAM when we apply both chunking and prefetching. TrackFM performs ~2.7× better than Fastswap for Sum, and ~2.9× better for Copy. In this case, Fastswap is limited by its page fault costs, and by its weaker ability to discern high-level knowledge about the access pattern. Note that AIFM could achieve similar (even slightly better) performance here, but would require programmer modifications.

4.4 Mitigating I/O Amplification

One of AIFM's major goals is to reduce I/O amplification, i.e., the unnecessary localization of unused memory, for workloads that access memory at fine granularity. Can TrackFM achieve the same goal? Figure 13a recreates our hashmap example, which will be sensitive to I/O amplification due to the small key/value pair sizes (4B). This time we show how overall performance is highly correlated with the amount of data transferred. We see how the smaller object size chosen by TrackFM significantly reduces the amount of data transferred over the network relative to Fastswap, which uses the standard 4KB page size. Fastswap transfers 43× the working set size for the hashmap, while TrackFM amplifies the working set by only 2.3× (the 64B object size chosen here is still larger than the key/value pairs). The net effect of reducing I/O amplification in this case is an average speedup of 12× relative to Fastswap. Though AIFM can achieve similar or higher speedups with programmer effort, this involves porting libc++ (457 KLOC) to AIFM, a non-trivial task.

Just the array storing the access trace for the keys requires 190MB, and with local memory constrained to 5% of total application memory (only 128 MB), we see high memory pressure, resulting in many object evacuations and swap-ins.

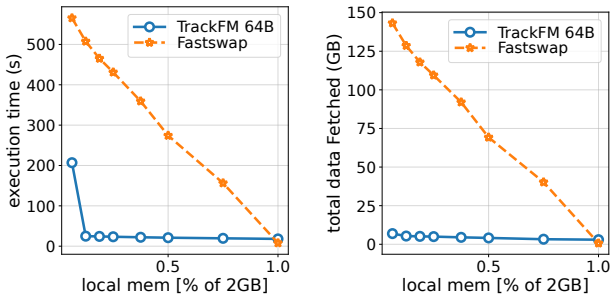


Figure 13. Applications that access memory at small granularities suffer when limited by the architected page size.

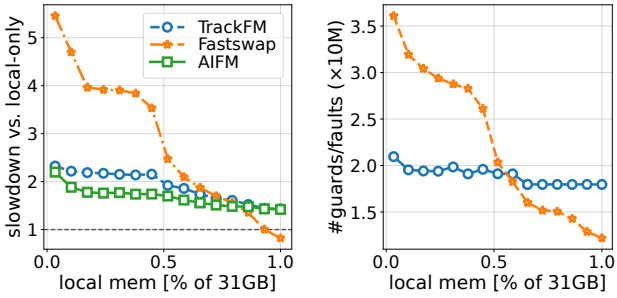


Figure 14. Performance of analytics application on TrackFM vs. Fastswap and AIFM. The left (a) shows overall performance normalized to a setup with only local memory, varying the amount of local memory available to the application. The right (b) shows the number of guard checks for TrackFM and page fault events for Fastswap. With less local memory, the page fault cost for Fastswap is amplified.

Thus, we see an inflated execution time (~200s) for the first point to the left of Figure 13a.

4.5 Application Benchmarks

How do injected guards, remote costs, and our optimizations translate to overall application performance? We explore this question with two application benchmarks. The first is a data analytics workload taken from Kaggle[12] that analyzes New York City taxi trips. We adapted this benchmark from AIFM to validate our results against that paper [35]. The second application is memcached [12], a commonly used in-memory key-value store. We also evaluate several benchmarks from the widely used NAS suite [5].

Analytics Application. The analytics application has a working set size of 31 GB. We compare the performance of

[12] https://www.kaggle.com/code/kartikkannapur/nyc-taxi-trips-exploratory-data-analysis/notebook

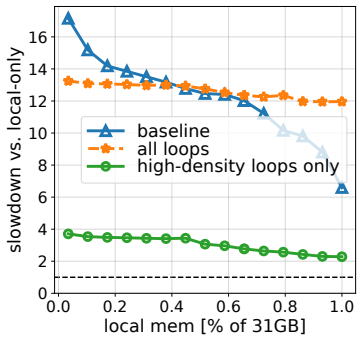


Figure 15. Applying the loop chunking optimization to low density objects in the analytics application reduces performance.

the application automatically transformed with TrackFM to the same application running on Fastswap and AIFM. This analytics application builds on a custom C++ dataframe library, and while we can correctly transform this library, our loop optimizations will not work efficiently due to C++ semantics such as exception handling, which the existing loop analysis in NOELLE has no support for. We concluded that supporting/extending NOELLE to support such C++-specific semantics would require engineering effort not justified by the research value added. Instead, we ported the original C++ dataframe library used in that paper to C, and the results reported for the analytics workload use the C dataframe library.

Figure 14 shows that when the available local memory is constrained, TrackFM comes within 10% of AIFM's performance, reaching near parity. Fastswap's performance converges when remote costs stop dominating, when roughly 75% of the working set fits in local memory. To explain these results, we measured the number of major (remote) page faults in Fastswap and the number of slow-path guards injected by TrackFM. We see that the page fault count (page faults imply one-sided RDMA operations in Fastswap) is relatively much higher than TrackFM guards; both event counts strongly correlate with overall performance. The analytics application consist of many column scan operations, which involve tight loops with almost no temporal locality but a high degree of spatial locality. TrackFM can exploit this to eliminate much of the guard costs, and also benefits from the AIFM runtime.

How impactful is our loop chunking optimization here? Figure 15 breaks down the performance similarly to Section 4.2, where we run the benchmark without loop chunking, with loop chunking applied to all loops, and with it applied only to candidate loops identified by our cost model. This application has several aggregation operations that involve loops that iterate over small collections of table rows (low object density), so applying the model here clearly has benefits for reducing guard costs.

Benchmark	Class	Memory (GB)	LoC
CG (conjugate gradient)	D	9	586
FT (3D FFT)	C	6	756
IS (bucket sort for integers)	D	34	558
MG (PDE solver with multigrid method)	D	27	941
SP (PDE solver with scalar penta-diagonal method)	D	12	2013

Table 3. NAS benchmarks (C++ versions) run on TrackFM.

Memcached. In-memory key-value stores represent another end of the access pattern spectrum. Here, access patterns tend to show much less spatial locality, and the granularity of access tends to be quite small, thus there is significant sensitivity to I/O amplification. We use TrackFM to automatically transform memcached version 1.2.7 to run as a far memory application. We use key/value pair sizes based on the USR distribution [4]. The working set size for memcached is 12GB, and we constrain the local memory to 1GB. We use a workload generator to create *get* operations on a Zipfian-distributed set of 100M keys. We measure the overall throughput for all get operations. Figure 16 shows the results. TrackFM shows a ~1.7× improvement over Fastswap when the skew parameter for the access distribution is between 1.01 and 1.04. As the access distribution becomes more skewed, we see an average speedup of 1.3× over Fastswap. As the skew parameter increases, Fastswap's performance converges due to increased temporal locality, which helps to amortize its page fault costs. While not shown in the figure, as we increase the amount of memory on the local node, Fastswap will converge with an even smaller skew, since more hot keys in the working set can fit on the local node. In this regime, TrackFM's fast-path guards become expensive, as they are not amortized like page faults. As the access distribution becomes less skewed, however, TrackFM outperforms Fastswap due to reductions in I/O amplification. We verify this by measuring the total data transferred over the network. Figure 16c shows that Fastswap, limited by the architected page size, transfers 66× the working set size, much of which is unnecessary since the key-value pair sizes are small. In contrast, TrackFM benefits from small object sizes and transfers only 15× the working set.

NAS benchmarks. We use a reference C++ implementation of the NAS serial benchmark suite [23], and select a limited subset (details shown in Table 3) due to time constraints. Figure 17a shows TrackFM outperforming Fastswap for most benchmarks, where page faults are the limiting factor for Fastswap. FT is a notable outlier where TrackFM performs poorly. First, the FFT implementation in NAS has a particularly friendly access pattern for Fastswap involving good temporal reuse, allowing it to amortize its page fault costs. Further investigation revealed that TrackFM is also injecting an exceptionally large number of guards for FT. We found that the deeply nested, tight loop structure used in FT confounds our loop analysis, resulting in the high guard

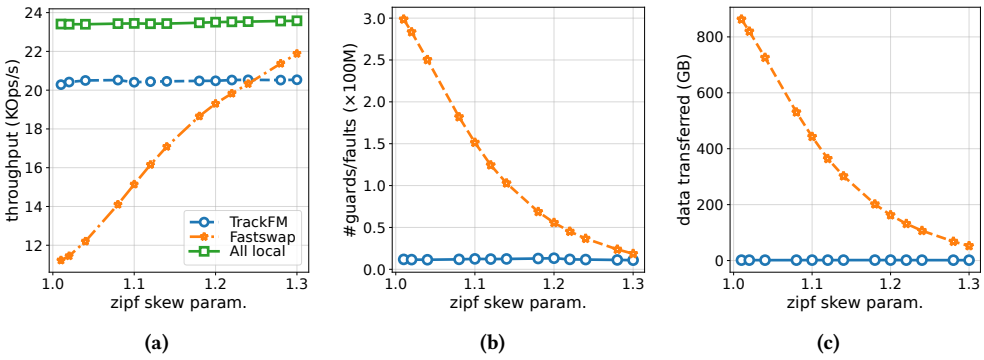


Figure 16. Key-value stores with small object sizes and little spatial locality suffer from I/O amplification in Fastswap.

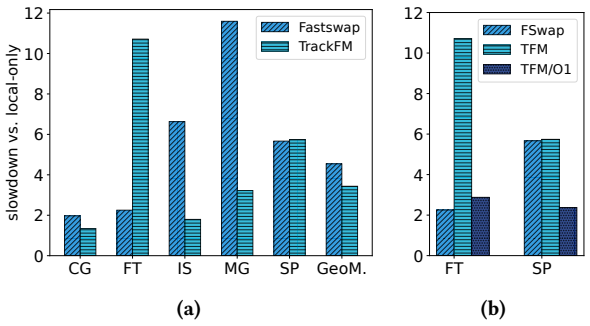


Figure 17. NAS benchmarks configured with a local memory size of 25% of the application working memory. Performance is normalized to local memory.

count. However, we found that this is mainly an artifact of the default analysis pipeline in NOELLE. By default, NOELLE sees unoptimized code from LLVM. However, in our case, it makes more sense to accept pre-optimized code in NOELLE to minimize the number of guards that are injected. For example, redundant code elimination or dead code elimination can reduce the number of loads and stores and thus the number of guards. We verified this in Figure 17b, where we perform the chain of optimizations included in the "O1" set *before* the TrackFM passes (TFM/O1). This results in a 6× reduction in memory instructions for FT, and a 4× reduction for SP, dramatically reducing guard overheads. This experiment led us to change NOELLE's default optimization pipeline order for use with TrackFM.

4.6 Compilation Costs

TrackFM increases generated code size by an average of 2.4× relative to the original binary. This increase is roughly proportional to the number of memory instructions in the program, each of which is expanded into a guard with the standard transformation. TrackFM's compile time is under 6× compared to standard LLVM, though we have not yet focused effort on reducing compilation overheads.

5 Discussion

Section 4 showed that the *compiler-based* approach holds promise. We now attempt to convey some hard-earned insights from our work, its limitations, and future prospects.

Lessons. We spent significant effort engineering the guards to be lightweight. This did pay off, but we were surprised to find that exploring ways to *eliminate* guards entirely was the more fruitful path, though this is somewhat obvious in retrospect. We were also surprised how well kernel-based approaches perform when there is sufficient *temporal* locality. This is because page fault costs are quickly amortized when there is repeated access. Even in this scenario, however, they are still sensitive to I/O amplification. This suggests that a hybrid approach (compiler and kernel) holds promise.

Understanding the high-level semantics of access patterns (i.e., access over an array, or a list, etc.) is critical for performance. We expect greater benefits when we can capture information about recursive data structures [25]. Finally, we found that in some cases, application code optimized for locality of reference can actually confound efforts by the compiler to derive fine-grained information about the access pattern. For example, memcached uses an optimized slab allocator that batches small allocations, thus grouping together small objects into large chunks. This actually limited TrackFM's ability to mitigate I/O amplification; TrackFM could have more effectively transformed this application had it performed small allocations in the naïve way.

Hardware Support. The overhead of TrackFM's guards could be improved with new hardware extensions. In the limit, the hardware can interpose on remote accesses and track dirty objects on its own, for example by extending the cache coherence engine (as in Kona [7]). However, while this approach is attractive from the standpoint of transparency, it forgoes the benefits of the high-level knowledge available

System	Programmer Transparent?	No custom hardware?	Mitigates I/O Amplification?	No OS Kernel Changes?
Project Kona [7]	✓	✗	✓	✗
AIFM [35]	✗	✓	✓	✓
Fastswap [3]	✓	✓	✗	✗
Infiniswap [13]	✓	✓	✗	✗
DiLOS [44, 45]	✓	✓	✓	✗
TrackFM (this work)	✓	✓	✓	✓

Table 4. Comparison of TrackFM with prior work.

to the compiler. An extension more appropriate for TrackFM might involve hardware that the compiler could manage, e.g., a lightweight, sub-page triggering mechanism that vectors *directly* to user-space (in contrast to the existing userfaultfd mechanisms in Linux [18]). This might, for example, look like a software/hardware stack built atop range translations [17] and user-level fault handling.[13]

Limitations and Future Work. The impact of AIFM's object size parameter is workload-dependent, so users must currently choose it. We believe it would be fairly simple to remove this engineering limitation by using autotuning, as discussed in Section 3.2.

Since TrackFM operates at the level of LLVM IR, information about application semantics (e.g., recursive data structures) is mostly lost. We plan to explore inter-procedural data structure analysis [22] to capture these semantics. There is also opportunity for languages whose memory semantics more closely match those of far memory, such as Rust, whose ownership model maps well to the notion of locality. High-level parallel languages, where ownership can fall out of language semantics [42], and partitioned global address space (PGAS) languages [9] could also map to compiler-based far memory.

Fetching remote data just to perform trivial computations is unwise. AIFM overcomes this by allowing library developers to manually offload such lightweight computations onto the remote node, thus employing *near-data processing*. We believe TrackFM could employ static analysis techniques, such as automated amortized resource analysis [15, 30], to achieve the same goal. TrackFM could also benefit from a profiling stage that prunes the set of heap allocations available for remoting based on access frequency. For example, the MaPHeA framework leverages hardware performance monitoring to enable profile-guided optimization (PGO) to effectively place heap-allocated objects in heterogeneous memory [33]. Though this framework is built on gcc, we suspect incorporating a similar approach into the TrackFM middle-end transformations would be straightforward.

[13] As in Intel's user-level interrupt vectoring introduced in the Sapphire Rapids microarchitecture [16].

6 Related Work

Prior work on far memory primarily falls along two lines: software and hardware-based. Hardware-based approaches center on the idea of removing the limitation of the architected page size [7, 14, 35]. On commodity machines, however, such specialized hardware is not yet an option. Prior work on improving *software-based*, programmer-transparent, far memory focuses on overcoming the limitations of the kernel-based approach, either by using better prefetching strategies [2, 6], by reducing page fault costs in the kernel [3], or by using high-performance networking [13]. Significant benefits are available when full programmer transparency is not a requirement, as shown by AIFM [35] and Carbink, which focuses on fault-tolerant far memory [47].

One way to improve on the kernel-based approach is to leverage a custom OS. DiLOS focuses on mitigating software overheads (especially of the paging subsystem) by building a LibOS specialized for disaggregated memory [44, 45]. DiLOS, which builds on OSv [19], uses a custom, *unified page table* that incorporates remote page table entries in lieu of repurposing the traditional swap cache to track remote page state, thus reducing software overheads. This approach can actually outperform AIFM with sufficient prefetching, demonstrating that in some cases reducing the page fault costs can counteract the negative effects of I/O amplification. However, even though DiLOS can run unmodified binaries (through POSIX compatibility), adopting a new OS can be a challenge. TrackFM, in contrast, runs on stock Linux without any changes.

Meta's production-scale far memory framework (TMO) leverages run-time information to transparently offload memory onto heterogeneous storage, and demonstrates that far memory pays off at scale [41].

Far memory systems share lineage with a large body of work on distributed shared memory (DSM), as these systems are similarly constrained by the architected page size. Thus, there is also work in this domain on avoiding page fault overheads. For example, Blizzard [37] and Shasta [36] work at sub-page granularity to mitigate false sharing. User-space approaches to DSM that leverage the compiler employ optimizations such as aggregation/hoisting of guards to reduce overheads [26, 28, 31]. However, these systems assume that an entire allocation is localized at once. In our system, chunks of a large allocation can be in independent states (local or remote), making hoisting optimizations more challenging. Prior approaches also assume that localized memory will not be evacuated again, which we must handle. Many of the optimizations applied in DSM systems relate to synchronization overheads and communication avoidance [8, 11, 24, 46], which are not applicable to non-coherent, far memory setups. TrackFM requires more careful analysis to reducing guard overheads since the same assumptions made for user-space DSM systems do not apply.

While unrelated to far memory, we build on ideas from prior work on using the compiler to replace paging-based address translation, namely CARAT [38] and CARAT CAKE [39]. Table 4 compares TrackFM to the most closely related work.

7 Conclusion

We demonstrated that the compiler-based approach to far memory is a feasible path to automatically transform applications to leverage remote memory. We realized the compiler-based approach with a prototype system called TrackFM, and demonstrated how it can outperform the kernel-based approach by up to 2× by merely recompiling the application. Its performance comes within 10% of the best performing library-based approach, AIFM, but requires no modifications to application code. TrackFM simultaneously achieves programmer transparency and good performance by leveraging novel compiler analysis and transformation techniques, and by using the highly-optimized AIFM runtime as a backend.

Acknowledgments

We thank Zhenyuan Ruan, Nicholas Wanninger, Tommy McMichen, Kevin McAfee, Enrico Deiana, Rolf Riesen, Balazs Gerofi, Matthew Wolf, and Ron Minnich for their assistance and discussions which helped make this paper possible. We also thank the anonymous reviewers and our shepherd, Marcos Aguilera, for their valuable feedback. This work was made possible with support from the United States National Science Foundation (NSF) via grants CCF-2028958, CNS-1763612, CNS-2239757, CNS-1763743, CCF-2028851, CCF-2119069, CCF-2107042, CNS-2211315, and CNS-2211508, the Department of Energy (DOE) via the Exascale Computing Project (17-SC-20-SC) and by the grant DE-SC0022268, as well as with generous support from Samsung Semiconductor, Inc. This work used resources from CloudLab [10], which is supported by the NSF's NSFCloud program.

References

[1] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2017. Remote Memory in the Age of Fast Networks. In *Proceedings of the Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 121–127. https://doi.org/10.1145/3127479.3131612

[2] Hasan Al Maruf and Mosharaf Chowdhury. 2020. Effectively Prefetching Remote Memory with Leap. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, USA, 843–857. https://www.usenix.org/conference/atc20/presentation/al-maruf

[3] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can Far Memory Improve Job Throughput?. In *Proceedings of the* 15^{th} *European Conference on Computer Systems* (Heraklion, Crete, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 14, 16 pages. https://doi.org/10.1145/3342195.3387522

[4] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-Scale Key-Value Store. *SIGMETRICS Performance Evaluation Review* 40, 1 (June 2012), 53–64. https://doi.org/10.1145/2318857.2254766

[5] David Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. 1995. *The NAS parallel benchmarks 2.0.* Technical Report NAS-95-020. NASA Ames Research Center. https://www.davidhbailey.com/dhbpapers/npb-2.0.pdf

[6] Christopher Branner-Augmon, Narek Galstyan, Sam Kumar, Emmanuel Amaro, Amy Ousterhout, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2022. 3PO: Programmed Far-Memory Prefetching for Oblivious Applications. arXiv:2207.07688 [cs.OS] https://arxiv.org/abs/2207.07688

[7] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. 2021. Rethinking Software Runtimes for Disaggregated Memory. In *Proceedings of the* 26^{th} *ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 79–92. https://doi.org/10.1145/3445814.3446713

[8] Michał Cierniak and Wei Li. 1995. Unifying Data and Control Transformations for Distributed Shared-Memory Machines. In *Proceedings of the 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation* (La Jolla, California, USA) *(PLDI '95)*. Association for Computing Machinery, New York, NY, USA, 205–217. https://doi.org/10.1145/207110.207145

[9] Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, François Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao, and Daniel Chavarría-Miranda. 2005. An Evaluation of Global Address Space Languages: Co-Array Fortran and Unified Parallel C. In *Proceedings of the* 10^{th} *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Chicago, IL, USA) *(PPoPP '05)*. Association for Computing Machinery, New York, NY, USA, 36–47. https://doi.org/10.1145/1065944.1065950

[10] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 1–14. https://www.usenix.org/conference/atc19/presentation/duplyakin

[11] Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel. 1996. An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System. In *Proceedings of the* 7^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, Massachusetts, USA) *(ASPLOS VII)*. Association for Computing Machinery, New York, NY, USA, 186–197. https://doi.org/10.1145/237090.237181

[12] Brad Fitzpatrick. 2004. Distributed caching with memcached. *Linux journal* 2004, 124 (2004), 5. https://www.linuxjournal.com/article/7451

[13] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *Proceedings of the* 14^{th} *USENIX Symposium on Networked Systems Design and Implementation* (Boston, MA) *(NSDI '17)*. USENIX Association, 649–667. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/gu

[14] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. 2022. Clio: A Hardware-Software Co-Designed Disaggregated Memory System. In *Proceedings of the* 27^{th} *ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 417–433. https://doi.org/10.1145/3503222.3507762

[15] Martin Hofmann and Steffen Jost. 2003. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New Orleans, Louisiana, USA) *(POPL '03)*. Association for Computing Machinery, New York, NY, USA, 185–197. https://doi.org/10.1145/604131.604148

[16] Intel Corporation 2023. *Intel® Architecture Instruction Set Extensions Programming Reference*. Intel Corporation. https://software.intel.com/content/www/us/en/develop/download/intel-architecture-instruction-set-extensions-programming-reference.html

[17] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. 2015. Redundant Memory Mappings for Fast Access to Large Memories. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. Association for Computing Machinery, New York, NY, USA, 66–78. https://doi.org/10.1145/2749469.2749471

[18] Linux Kernel. [n. d.]. Userfaultfd. https://www.kernel.org/doc/Documentation/vm/userfaultfd.txt

[19] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. 2014. OSv—Optimizing the Operating System for Virtual Machines. In *Proceedings of the 2014 USENIX Annual Technical Conference* (Philadelphia, PA) *(USENIX ATC '14)*. USENIX Association, 61–72. https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity

[20] 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. 2019. Software-Defined Far Memory in Warehouse-Scale Computers. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 317–330. https://doi.org/10.1145/3297858.3304053

[21] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization* (San Jose, CA, USA) *(CGO '04)*. IEEE, 75–86. https://doi.org/10.1109/CGO.2004.1281665

[22] Chris Lattner and Vikram Adve. 2005. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 129–142. https://doi.org/10.1145/1065010.1065027

[23] Júnior Löff, Dalvan Griebler, Gabriele Mencagli, Gabriell Araujo, Massimo Torquati, Marco Danelutto, and Luiz Gustavo Fernandes. 2021. The NAS Parallel Benchmarks for evaluating C++ parallel programming frameworks on shared-memory architectures. *Future Generation Computer Systems* 125 (Dec. 2021), 743–757. https://doi.org/10.1016/j.future.2021.07.021

[24] Honghui Lu, Alan L. Cox, Sandhya Dwarkadas, Ramakrishnan Rajamony, and Willy Zwaenepoel. 1997. Compiler and Software Distributed Shared Memory Support for Irregular Applications. In *Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Las Vegas, Nevada, USA) *(PPoPP '97)*. Association for Computing Machinery, New York, NY, USA, 48–56. https://doi.org/10.1145/263764.263772

[25] Chi-Keung Luk and Todd C. Mowry. 1996. Compiler-Based Prefetching for Recursive Data Structures. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, Massachusetts, USA) *(ASPLOS VII)*. Association for Computing Machinery, New York, NY, USA, 222–233. https://doi.org/10.1145/237090.237190

[26] Govindarajan R. Manoj N. P., Manjunath K. V. 2004. CAS-DSM: A Compiler Assisted Software Distributed Shared Memory. *International Journal of Parallel Programming* 32, 2 (2004), 77–122.

[27] Angelo Matni, Enrico Armenio Deiana, Yian Su, Lukas Gross, Souradip Ghosh, Sotiris Apostolakis, Ziyang Xu, Zujun Tan, Ishita Chaturvedi, Brian Homerding, et al. 2022. NOELLE Offers Empowering LLVM Extensions. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization* (Seoul, Republic of Korea) *(CGO '22)*. IEEE, 179–192. https://doi.org/10.1109/CGO53902.2022.9741276

[28] Takashi Matsumoto and Kei Hiraki. 1997. Memory-based communication facilities and asymmetric distributed shared memory. In *Proceedings of the 1st International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems* (Maui, HI, USA) *(IWIA '97)*. IEEE, 30–39. https://doi.org/10.1109/IWIA.1997.670405

[29] John D. McCalpin. 1991–2007. *STREAM: Sustainable memory bandwidth in high performance computers*. Technical Report. University of Virginia.

[30] Stefan K. Muller and Jan Hoffmann. 2021. Modeling and Analyzing Evaluation Cost of CUDA Kernels. *Proceedings of the ACM on Programming Languages* 5, POPL, Article 25 (Jan. 2021), 31 pages. https://doi.org/10.1145/3434306

[31] Junpei Niwa, Tatsushi Inagaki, Takashi Matsumoto, and Kei Hiraki. 1999. Evaluation of Compiler-Assisted Software DSM Schemes for a Workstation Cluster. In *Proceedings of the 3rd International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems* (Maui, HI, USA) *(IWIA '99)*. IEEE, 57–68. https://doi.org/10.1109/IWIA.1999.898843

[32] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2014. Scale-out NUMA. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 3–18. https://doi.org/10.1145/2541940.2541965

[33] Deok-Jae Oh, Yaebin Moon, Do Kyu Ham, Tae Jun Ham, Yongjun Park, Jae W. Lee, Jung Ho Ahn, and Eojin Lee. 2022. MaPHeA: A Framework for Lightweight Memory Hierarchy-Aware Profile-Guided Heap Allocation. *ACM Transactions on Embedded Computing Systems* 22, 1 (Dec. 2022), 1–28. https://doi.org/10.1145/3527853

[34] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-Sensitive Datacenter Workloads. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI '19)*. USENIX Association, Berkeley, CA, USA, 361–377. https://doi.org/10.5555/3323234.3323265

[35] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. 2020. AIFM: High-Performance, Application-Integrated Far Memory. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation* (Virtual, USA) *(OSDI '20)*. USENIX Association, Berkeley, CA, USA, 315–332. https://www.usenix.org/conference/osdi20/presentation/ruan

[36] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. 1996. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proceedings of the 7th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, Massachusetts, USA) *(ASPLOS VII)*. Association for Computing Machinery, New York, NY, USA, 174–185. https://doi.org/10.1145/237090.237179

[37] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. 1994. Fine-Grain Access Control for Distributed Shared Memory. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS VI)*. Association for Computing Machinery, New York, NY, USA, 297–306. https://doi.org/10.1145/195473.195575

[38] Brian Suchy, Simone Campanoni, Nikos Hardavellas, and Peter Dinda. 2020. CARAT: A Case for Virtual Memory through Compiler- and Runtime-Based Address Translation. In *Proceedings of the* 41^{st} *ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI '20)*. Association for Computing Machinery, New York, NY, USA, 329–345. https://doi.org/10.1145/3385412.3385987

[39] Brian Suchy, Souradip Ghosh, Drew Kersnar, Siyuan Chai, Zhen Huang, Aaron Nelson, Michael Cuevas, Alex Bernat, Gaurav Chaudhary, Nikos Hardavellas, Simone Campanoni, and Peter Dinda. 2022. CARAT CAKE: Replacing Paging via Compiler/Kernel Cooperation. In *Proceedings of the* 27^{th} *ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 98–114. https://doi.org/10.1145/3503222.3507771

[40] Muhammad Tirmazi, Adam Barker, Nan Deng, Md Ehtesam Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: the Next Generation. In *Proceedings of the* 15^{th} *European Conference on Computer Systems* (Herakleion, Crete, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 30, 14 pages. https://doi.org/10.1145/3342195.3387517

[41] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. 2022. TMO: Transparent Memory Offloading in Datacenters. In *Proceedings of the* 27^{th} *ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 609–621. https://doi.org/10.1145/3503222.3507731

[42] Michael Wilkins, Sam Westrick, Vijay Kandiah, Alex Bernat, Brian Suchy, Enrico Armenio Deiana, Simone Campanoni, Umut Acar, Peter Dinda, and Nikos Hardavellas. 2023. WARDen: Specializing Cache Coherence for High-level Parallel Languages. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization* (Montréal, QC, Canada) *(CGO '23)*. Association for Computing Machinery, New York, NY, USA, 122–135. https://doi.org/10.1145/3579990.3580013

[43] Michael E. Wolf and Monica S. Lam. 1991. A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '91)*. Association for Computing Machinery, New York, NY, USA, 30–44. https://doi.org/10.1145/113445.113449

[44] Wonsup Yoon, Jinyoung Oh, Jisu Ok, Sue Moon, and Youngjin Kwon. 2021. DiLOS: Adding Performance to Paging-Based Memory Disaggregation. In *Proceedings of the* 12^{th} *ACM SIGOPS Asia-Pacific Workshop on Systems* (Hong Kong, China) *(APSys '21)*. Association for Computing Machinery, New York, NY, USA, 70–78. https://doi.org/10.1145/3476886.3477507

[45] Wonsup Yoon, Jisu Ok, Jinyoung Oh, Sue Moon, and Youngjin Kwon. 2023. DiLOS: Do Not Trade Compatibility for Performance in Memory Disaggregation. In *Proceedings of the* 18^{th} *European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 266–282. https://doi.org/10.1145/3552326.3567488

[46] Matthew J. Zekauskas, Wayne A. Sawdon, and Brian N. Bershad. 1994. Software Write Detection for Distributed Shared Memory. In *Proceedings of the* 1^{st} *Symposium on Operating Systems Design and Implementation* (Monterey, CA, USA) *(OSDI '94)*. USENIX Association, 8–es. https://www.usenix.org/conference/osdi-94/software-write-detection-distributed-shared-memory

[47] 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. 2022. Carbink: Fault-Tolerant Far Memory. In *Proceedings of the* 16^{th} *USENIX Symposium on Operating Systems Design and Implementation (OSDI '22)*. USENIX Association, Carlsbad, CA, 55–71. https://www.usenix.org/conference/osdi22/presentation/zhou-yang

A Artifact Appendix

A.1 Abstract

The artifact comprises our TrackFM runtime, compiler passes, experiment scripts and data sets. In our github repository we have detailed instructions on how to run TrackFM.

A.2 Artifact check-list (meta-information)

- **How much time is needed to prepare workflow (approximately)?: 3 hours**
- **How much time is needed to complete experiments (approximately)?: 12 days**
- **Publicly available? : https://github.com/compiler-disagg/TrackFM**
- **Code licenses (if publicly available)?: MIT**

A.3 Description

A.3.1 How to access. All code for TrackFM can be found at https://github.com/compiler-disagg/TrackFM. The detailed instructions on how to run TrackFM are provided in the `README.md` file located in the top level of the repository.

A.3.2 Hardware dependencies. To run these experiments, we require two ten-core Intel E5-2640v4, 64GB Memory, 300GB Disk and Mellanox ConnectX-4 25 GB NIC. We recommend CloudLab to run our experiments.

A.3.3 Software dependencies. TrackFM mainly relies on llvm-9.0.0 and noelle-v9.8.0 the compilation passes. We also have dependencies on OS packages, specific Linux kernel versions, and NIC drivers to run the experiments and generate plots. These details are provided in the TrackFM repository.

A.3.4 Data sets. We host our datasets in Kaggle and provide detailed instructions in the repository on how to use them.

A.4 Installation

The TrackFM runtime can be installed by running the `./build.sh` script located in the `TrackFM/runtime` directory. TrackFM compilation passes can be installed by running `make -j` in the `TrackFM/runtime/compiler_passes/passes` directory. Once TrackFM is setup, one can verify the installation by running `make smoke_test` from the root directory. More details are provided in our README.

A.5 Experiment workflow

All experiments can be run from the top-level directory of the code repo using *make*. For example, to reproduce Figure 14a, one can run `make trackfm_fig14a` from the root directory.

A.6 Evaluation and expected results

A.6.1 Major claims.

- (C1) Loop Chunking eliminates fast-path guards and improves speedup for STREAM benchmarks. We show this in experiment E1 described in Section 4.2, whose results are shown in Figure 7.
- (C2) Loop chunking benefits from avoiding collections with low object density. We show this in experiment E2 described in Section 4.2, whose results are depicted in Figure 8.
- (C3) Fine-grained memory accesses with little spatial locality can benefit from small object sizes. We show this in experiment E3 described in Section 4.3 with results shown in Figure 9.
- (C4) Access patterns with high spatial locality benefit from the choice of a larger object size. This is shown in experiment E4 described in Section 4.3, and the results are in Figure 10.
- (C5) Loop chunking coupled with prefetching helps TrackFM extract more performance from workloads with spatial locality. We show this in experiment E5 described in Section 4.3; results are shown in Figure 11.
- (C6) TrackFM 's memory analysis helps to best exploit AIFM's high-performance prefetching. We show this in experiment E6 in Section 4.3, depicted in Figure 12.
- (C7) Applications that access memory at small granularities suffer when limited by the architected page size. This is shown in experiment E7 described in Section 4.4, whose results are shown in Figure 13.
- (C8) With less local memory, TrackFM outperforms Fastswap. We show this in experiment E8 in Section 4.5, with results in Figure 14.
- (C9) Applying the loop chunking optimization to low-density objects in the analytics application reduces performance, shown in experiment E9 described in Section 4.5, with results in Figure 15.
- (C10) Key-value stores with small object sizes and little spatial locality suffer from I/O amplification in Fastswap. We show this in experiment E10, Section 4.5, and Figure 16.
- (C11) With a 25% local memory constraint, the NAS benchmarks benefit from TrackFM. We show this in experiment E11 in Section 4.5, with results in Figure 17a.

A.6.2 Experiments. All experiments can be reproduced using `make`. On completion, `make` will generate a figure in the `figs` directory located at the top level of the repository. This figure can then be compared with the respective figure in the paper to evaluate our claims.

Experiment (E1). This experiment helps to evaluate claim C1.

[How to]. `make_trackfm_fig7`

[Results]. A graph named `fig7.png` will be generated in `figs` directory. You should be able to see the benefits of loop chunking.

Experiment (E2). This experiment helps to validate claim C2.

[How to]. `make_trackfm_fig8`

[Results]. A graph named `fig8.png` will be generated in the `figs` directory. You should be able to see the importance of applying loop chunking selectively in TrackFM.

Experiment (E3). This experiment helps to evaluate claim C3.

[How to]. `make_trackfm_fig9`

[Results]. A graph named `fig9.png` will be generated in the `figs` directory. You will see the benefits of small object sizes for applications with irregular access patterns.

Experiment (E4). This experiment helps to validate claim C4.

[How to]. `make_trackfm_fig10`

[Results]. A graph named `fig10.png` will be generated in the `figs` directory. You will see that large object sizes improve performance for applications with spatial locality.

Experiment (E5). This experiment helps to validate claim C5.

[How to]. `make_trackfm_fig11`

[Results]. A graph named `fig11.png` will be generated in the `figs` directory. You will see the benefits of prefetching combined with loop chunking in TrackFM.

Experiment (E6). This experiment helps to validate claim C6.

[How to]. `make_trackfm_fig12`

[Results]. A graph named `fig12.png` will be generated in the `figs` directory. You will see that TrackFM can be 2-3*times* better than Fastswap when it understands the application's access patterns.

Experiment (E7). This experiment helps to validate claim C7.

[How to]. `make_trackfm_fig13`

[Results]. A graph named `fig13.png` will be generated in `figs` directory. You will see that TrackFM can mitigate I/O amplification.

Experiment (E8). This experiment helps to validate claim C8.

[How to]. `make_trackfm_fig14a`

[Results]. A graph named `fig14a.png` will be generated in the `figs` directory. You will see that TrackFM is within 10% of AIFM's performance.

Experiment (E9). This experiment helps to validate claim C9.

[How to]. `make_trackfm_fig15`

[Results]. A graph named `fig15.png` will be generated in the `figs` directory. You will see that selectively applying loop chunking is critical to applications which have low object density.

Experiment (E10). This experiment helps to validate claim C10.

[How to]. `make_trackfm_fig16a`

[Results]. A graph named `fig16a.png` will be generated in the `figs` directory. You will see that even when TrackFM optimizations do not apply, it still benefits from the use of small object sizes.

Experiment (E11). This experiment helps to validate claim C11.

[How to]. `make_trackfm_fig17a`

[Results]. A graph named `fig17a.png` will be generated in `figs` directory. You will see that NAS benchmarks benefit from TrackFM when less memory is available.

A.7 Notes on Reusability

We provide several `make` scripts to automate new applications with TrackFM. We provide instructions on how to use TrackFM for new applications in the `README` in the top level of the TrackFM repository.