Pre-Stores: Proactive Software-guided Movement of Data Down the Memory Hierarchy

Xiaoxiang Wu
University of Sydney
Sydney, Australia

Baptiste Lepers
Inria, Grenoble, France
University of Neuchâtel
Neuchâtel, Switzerland

Willy Zwaenepoel
University of Sydney
Sydney, Australia

Abstract

We introduce the notion of software pre-storing – the converse of software pre-fetching. With software pre-fetching, instructions are inserted in the code to asynchronously move data up in the memory hierarchy. With software pre-storing, instructions are inserted to direct the CPU to asynchronously move data *down* in the memory hierarchy. Pre-storing can be implemented by using existing processor instructions.

Software pre-storing provides performance benefits for write-heavy applications, especially with emerging architectures that incorporate memories with diverse characteristics such as, for instance, remote DRAM accessed via a CXL switch or nonvolatile PMEM memory. We identify application scenarios in which software pre-storing is beneficial, and we have developed a tool, DirtBuster, that identifies applications and code regions that can benefit from pre-storing.

We evaluate the concept of software pre-storing and the DirtBuster tool on two CPU architectures (ARM and x86) and two types of cacheable memories (PMEM and cache-coherent DRAM accessed through an FPGA). We demonstrate performance improvements for key-value stores, HPC applications, message passing, and Tensorflow, by up to 2.3×.

CCS Concepts: • **Software and its engineering → Main memory**; • **Computer systems organization → Heterogeneous (hybrid) systems**.

Keywords: CPU caches, pre-fetch, pre-store, hybrid memory

ACM Reference Format:
Xiaoxiang Wu, Baptiste Lepers, and Willy Zwaenepoel. 2025. Pre-Stores: Proactive Software-guided Movement of Data Down the Memory Hierarchy . In *Proceedings of Twentieth European Conference on Computer Systems (EuroSys '25).* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3689031.3696097

EuroSys '25, March 30–April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1196-1/25/03
https://doi.org/10.1145/3689031.3696097

1 Introduction

In newer memory architectures, CPU caches store data from memories with a growing range of access characteristics, such as differences in bandwidth, latency or size of the transfer unit between cache and memory. For instance, recent Intel CPUs can cache data from standard DDR memory, but also from high bandwidth memory (HBM) [13], persistent memory (PMEM) [10] and CXL-attached storage [11]. Furthermore, an increasing number of NUMA servers comprise standard ARM or x86 nodes interconnected in a cache-coherent manner with an accelerator or an FPGA [9].

Caches are designed with conventional DRAM in mind. We demonstrate that they perform suboptimally when they cache data for memories with characteristics substantially different from DRAM. Specifically, differences between the sizes of the cache lines and and the memory unit cause write amplification. Furthermore, high latencies can cause memory coherence operations to be delayed.

We show that cache performance can be improved by asynchronously moving data down the memory hierarchy "earlier" than would be necessary because of memory models or resource constraints. We formalize this idea with the concept of software *pre-storing* (simply referred to as pre-storing in the rest of the paper). A pre-store is the converse of a pre-fetch. While a pre-fetch directs the CPU to move data up the memory hierarchy [21, 27, 30, 33–35, 37, 46], a pre-store directs the CPU to move data down the memory hierarchy. We describe different forms of pre-storing, including moving data from private CPU storage to globally visible cache locations and moving data from the cache to memory.

We present application scenarios in which pre-stores provide performance benefits. First, consider a scenario in which one or more writes are followed sometime later by a fence. The CPU is allowed to keep the data written in its registers or in a private buffer, outside of the cache, until the occurrence of the fence. When hitting the fence, the CPU sends the writes to the cache, the cache reads the corresponding cache lines from memory if necessary, and updates them. These cache line reads and updates must be completed before the fence can complete. Current cache implementations have been optimized to hide the latency of conventional DRAM, but fail to hide the latency of less conventional memories. Pre-stores direct the CPU to asynchronously initiate

the writes to the CPU caches in advance, with the goal that they complete before the fence is reached.

Second, performance issues also arise when the CPU evicts data from its caches. Consider a scenario in which an application writes contiguous data items, e.g., successive elements in an array. Data is written to the cache sequentially and eventually written back to memory, but because of unpredictable cache conflicts or non-LRU cache replacement, writebacks to memory may not be sequential. The lack of sequentiality in evictions is typically not a problem for DRAM, but can negatively impact memories with large write granularities (e.g., Optane Persistent memory or CXL-based storage that internally write data in chunks of 256B or more). Pre-stores improve sequentiality of writes from the cache to memory.

We describe DirtBuster, a tool that identifies possible performance anomalies of the type discussed above, suggests locations where a pre-store can be inserted, and what type of pre-store would be most beneficial. DirtBuster relies on dynamic analysis, using a combination of memory access sampling and binary instrumentation.

We present measurement results of four sets of applications: a subset of the Phoronix benchmarks [32], including TensorFlow [1], a subset of the NAS benchmarks [5], two key-value stores [16, 31] and a message passing benchmark. We demonstrate the benefits of pre-storing on different CPU architectures and cacheable memories. Pre-storing improves performance by up to a factor of 2.3×.

The contributions of this paper are:

- The identification of performance issues with the interaction between caches and (unconventional) memory systems.
- The introduction of the concept of software pre-storing.
- A tool for discovering code segments in which software pre-storing can address the aforementioned performance issues.
- An evaluation of pre-stores to mitigate the observed performance problems.

The outline of the rest of this paper is as follows. Section 2 introduces the concept of pre-storing. Section 3 introduces the diversity of architectures and cacheable memories. Section 4 provides use cases for pre-stores. Section 5 discusses the possible overheads of using pre-stores. Section 6 presents the design and implementation of DirtBuster. Section 7 evaluates the impact of pre-stores. Section 8 presents the related work, and Section 9 concludes.

2 Pre-stores

We provide a function that allows moving data down the memory hierarchy:

```
prestore(void *location, size_t size, op_t op)
```

The `op` parameter is used to indicate the desired operation. It can take the following values:

- `demote`. A `demote` pre-store moves data down in the cache hierarchy, for instance, from the L1 cache to the L2 cache, or from private CPU buffers to the L1 cache
- `clean`. A `clean` pre-store directs the CPU to write dirty data from the cache to memory.

The pre-store operation acts on `size` bytes, starting from `location`. Regardless of the `op` flag, pre-stores keep the data in the cache: *cleaning* the data propagates the modifications to memory but does not invalidate the cache. The `prestore` function is non-blocking: data is moved down the cache hierarchy or written to memory in the background without blocking the CPU pipeline.

It is also possible to *skip* the cache entirely, writing a value directly from registers to memory with so-called non-temporal stores. Unlike *demotion* and *cleaning*, which can be achieved by simply inserting a suitable call to `prestore`, non-temporal stores require modifying the code in a more complicated manner.

Implementation of pre-stores. Common architectures such as x86 and ARM offer instructions that allow easy implementation of pre-stores.

Intel CPUs allow developers to move data down the cache hierarchy with the `cldemote` instruction (*demotion* operation). Data can also be moved from caches to memory using the `clwb` instruction (cache line writeback, or *cleaning*).

ARM CPUs offer a wide range of cache-related instructions. For instance, `dc cvau` directs the CPU to write data to the "point of unification", the point at which all cores in the system see the same value (the L2 cache for most modern devices) [43].

Regardless of the architectures, all the instructions presented in this section are non-blocking, allowing for a non-blocking implementation of pre-store. In previous works, these instructions have typically been used to force the CPU to write data in a specified order, for instance, to provide a consistent representation of data to all cores, or to persist data in order. In our work, we use these instructions to direct the CPU to execute writes in the background.

3 The diversity of memory architectures

The classical memory architecture consists of a number of levels of cache and a single type of memory (DRAM). Given the single type of memory sitting beneath the caches, the cache design is closely integrated with the memory system.

Recently, architectures have come on the market that deviate from this classical design. Multiple types of memory can be cached by a single system of caches. In this paper, we focus on two characteristics that we believe will become commonplace in future servers.

Differences between the CPU cache line size and the internal write granularity of the memory. These differences are likely to become the norm, as caches are asked to

cache more and more diverse devices. Table 1 presents the internal granularity of reads and writes for different CPUs and memory devices. For instance, storage uses cache lines size $4-8\times$ larger than that of Intel CPUs.

Table 1. Devices internally read and write at different granularities (e.g., 64B cache line for an Intel CPU vs. 256B for an Optane persistent memory).

Device	Internal granularity
Intel CPU	64B
ThunderX ARM CPU	128B
Optane PMEM	256B
CXL SSD	256B/512B (current technologies [41])

As a currently available example of this type of architecture, we use a two-node NUMA machine, with 40 Intel Xeon Gold 6230 cores running at 2.10GHz, 128GB of DRAM, and 8*128GB Intel Optane NV-DIMMs. In this machine, the CPU caches data at a 64B granularity, but the Optane memory internally writes data at a 256B granularity. We refer to this configuration as *Machine A*.

Weak memory architectures and long latency memories. With the current trend towards architectures with disaggregated memory, caches are required to front memories with longer latencies than directly attached DRAM. Byte-addressable memory accessible over CXL [39] is a prime example of such architectures, but other examples include accelerators for AI, streaming and networking workloads [2, 12, 20] and cache-coherent FPGAs [9].

As a currently available instance of this type of architecture, we use an Enzian [9] prototype, referred to as *Machine B*. Enzian is an asymmetric NUMA system with a 48-core ArmV8 ThunderX-1 and a Xilinx XCVU9P Ultrascale+ FPGA. The FPGA's memory is transparently cached by the CPU. Figure 1 depicts the topology of machine B. The applications run on the CPU, which transparently caches the FPGA's memory.

The latency and bandwidth of the FPGA can be configured. To illustrate the diversity of devices that will be accessed in a cache-coherent manner, we test two configurations of machine B: a lower-latency configuration, in which the FPGA's memory is accessed in 60 cycles at 10GB/s (representative of future high-end CXL-accessible memory), and a higher-latency configuration in which the FPGA is accessed in 200 cycles at 1.5GB/s (representative of medium-tier CXL-accessible storage). We refer to these two configurations as *Machine B-Fast* and *Machine B-Slow*, respectively.

4 Example uses of pre-stores

This section provides simple examples of the use of pre-stores and measurements to illustrate their benefits.

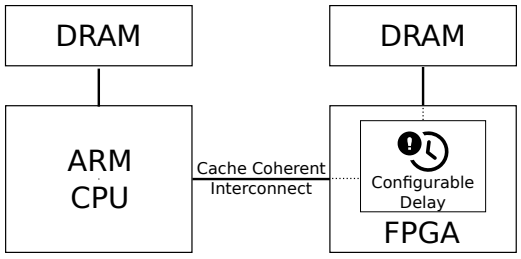

Figure 1. Topology of Machine B.

4.1 Problem #1 - The random order of evictions (on Machine-A)

Modern CPU caches tend to evict data in a seemingly random order. Even when data is written sequentially by the application, the corresponding cache lines may not be written out in the same order to memory. We explain why this behavior impacts performance.

Context. CPU caches are associative: the cache is divided into bins, and each bin can hold n cache lines (n-way associative cache). Replacement in a bin is often modeled by simple LRU policy, but modern caches rely on much more complex strategies [40]. For instance, Intel CPUs rely on a pseudo-LRU and "random" evictions to reduce the cost of maintaining LRU [45]. Similarly, ARM CPUs implement a mix of LRU, FIFO, and random evictions [3].

Figure 2 illustrates the state of a 2-way cache, writing four arrays, one after the other. The first two arrays fit in the cache, and no conflict occurs (cache at t_0). When writing the third array, the cache needs to evict data. In strict LRU order, the cache would evict the first array and replace it with the third, but in reality the cache evicts data from both the first and the second array (cache at t_1). When the fourth array is written, data from the three previously written arrays is evicted (cache at t_2).

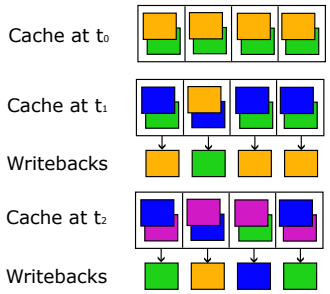

Figure 2. Content of a 2-way cache after writing four arrays, one after the other. Each array is presented in a different color. Because the cache evicts data in "random" order, a given array may be partially evicted multiple times.

The problem of the random order of evictions is worsened when executing multiple threads on multiple cores. The interleaving of the memory accesses performed by the threads results in seemingly random memory accesses at the Last Level Cache (LLC) and decreases the likelihood of the cache evicting data sequentially.

Impact. Non-sequential evictions are problematic when the size of the write unit of the cached medium is different from the CPU's cache line size.

For instance, on machine A, the Intel CPUs evict 64B cache lines, but the internal granularity of Optane Persistent memory is 256B. In the example of Figure 2, the first writeback at t_1 flushes four cache lines (for a total of 256B) to persistent memory, but internally the device writes $2 \times 256B$: 256B for the 3 cache lines of the yellow array and 256B for the 1 cache line of the blue array, resulting in a 2× write amplification. The second writeback at t_2 results in a 3× write amplification.

Guiding the cache. Instead of relying on the hardware-defined order of evictions, it is possible to use *clean* pre-stores to approximate sequential cache eviction. For instance, in the example of Figure 2, it is possible to ask the CPU to clean the first array (yellow cache lines) before reading the third array (blue cache lines). The first array is then very likely written to memory sequentially, and the reading of the third array causes no write amplification.

Performance. Listing 1 presents a simple example to evaluate the impact of pre-stores on performance. Multiple threads write elements of an array in random order. The elements are then re-read to compute a total sum. We vary the size of the elements from 64B (simulating a succession of small random writes) to 4KB (simulating a succession of sequential writes). We evaluate the impact of pre-storing the elements just after their initialization.

Listing 1 Simple example to illustrate the impact of pre-storing data.

```
1  parallel_for(...) {
2   size_t idx = rand() % nb_elements;
3   memcpy(&elts[idx], ..., <sizeof elt>);
4   prestore(&elts[idx], <sizeof elt>, clean);
5   total += elt[idx].field;
6  }
```

Figure 3(a) presents the performance improvement brought about by pre-stores, and Figure 3(b) shows the write amplification when running Listing 1 in persistent memory. Adding pre-stores results in reduced write amplification and in up to a 3× performance improvement.

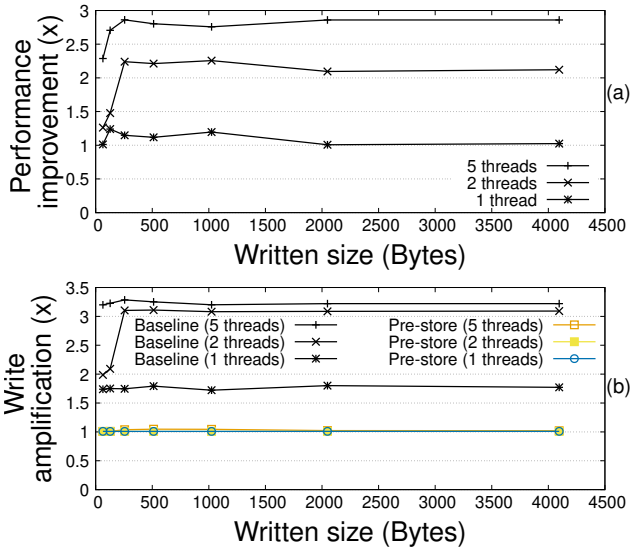

Figure 3. Machine A. (a) Improvement brought about by activating the *cleaning* pre-store call in Listing 1, varying the element size. (b) Write amplification with and without *cleaning*.

We measure the write amplification by comparing the number of 64B cache lines evicted from the cache to the amount of data actually written (both numbers are collected using the `ipmctl` [15] tool). If it were the case that the cache was evicting data in the order it was written, pre-storing the elements would have no impact on performance when using large item sizes (e.g., with 4KB items, the cache would evict data in 4KB sequential chunks and would maximize PMEM bandwidth). When using 1 thread, without pre-storing, however, the random order of evictions causes a 180% write amplification in persistent memory (every 64B cache line writeback results in 115B written in persistent memory). When using two or more threads, the write amplification grows to 330% (every 64B writeback results in 211B written in persistent memory), close to the maximum 400% write amplification for persistent memory. Pre-stores eliminate write amplification entirely.

The impact of eliminating write amplification on performance depends on the contention on the cached medium. With a single thread, the persistent memory is far from saturated, and the internal write amplification does not impact performance. With more than 2 threads, the write amplification limits the available bandwidth, and pre-stores improve performance by 2.2× (two threads), and up to 3× (five threads).

Summary. It is possible to direct the cache to write dirty data sequentially to memory. Sequentiality improves performance, especially when the cache line size of the CPU and the internal write granularity of the cached device differ.

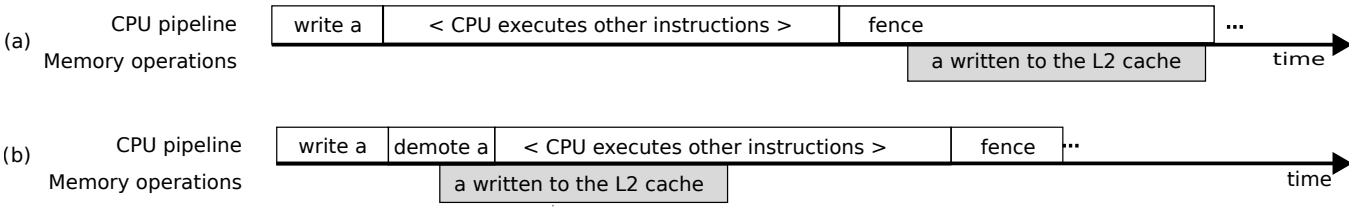

Figure 4. Demoting data forces the CPU to write the data to its caches. Without the demote instruction, the CPU can keep the modified a private, until it is forced to commit the change, causing delays in the execution of the CPU pipeline.

4.2 Problem #2 - Delayed cache operations (on Machine-B)

We explain why delayed cache operations can negatively impact performance.

Context. When writing data, CPUs are allowed to keep the changes private, as long as the changes do not break the memory ordering constraints of the architecture. Because cache coherence operations are expensive, CPUs tend to keep modifications private and only advertise them when they run out of private buffer space or when they are forced to by the memory model.

Impact. Figure 4(a) illustrates the impact of keeping modifications private. The CPU executes a write instruction, followed by other instructions, followed by a memory fence. The reads and write prior to the fence must occur before the reads and writes placed after the fence. For brevity, in this paper we refer to all instructions that implement the fence semantics simply as "fences". Other such instructions include, for instance, atomic instructions.

A consequence of the fence is that the CPU has to make all prior writes public before executing any other memory instruction. If the CPU has private modifications, publicizing these modifications happens "at the last minute", while executing the fence, and stalls the CPU pipeline.

Guiding the cache. It is possible to force CPUs to write data to the cache, without stalling, using cache demoting instructions (*demote* operation of pre-store). Figure 4(b) illustrates the use of a demote pre-store. By issuing the pre-store, the application demotes the data, which triggers the data to be written to the cache in the background.

Impact on performance. Listing 2 presents the pseudo-code of the example we use to assess the performance impact of delayed write operations. The example writes a cache line (128B on Machine B), demotes it, and then performs a configurable number of reads to its L1 cache. These operations are followed by a fence. We repeat the write-prestore-read-fence sequence 10 million times and measure the total runtime.

Listing 2 Pseudo-code to illustrate the performance impact of writes before a fence.

```
1  while(...) {
2      size_t idx = rand() % num_elements;
3      memset(&array[idx], ..., 128);
4      prestore(&array[idx], 128, demote);
5      for(int i = 0; i < n; i++)
6         read(&L1_data[i]);
7      fence(); // could also be an atomic op
8  }
```

Figure 5 presents the impact of the demotion on performance. We observe that demoting data is up to 65% faster than not demoting data on machine B.

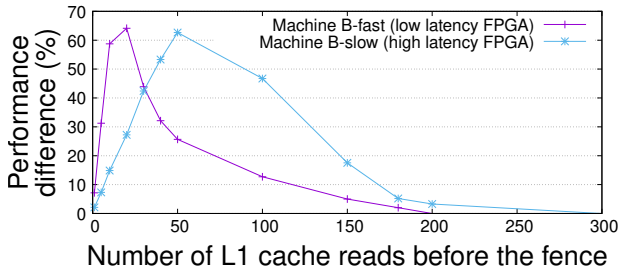

Figure 5. Machine B executing Listing 2. Relative performance improvement brought by demoting the dirty data before the fence. Demotion improves performance by up to 65%. The higher the latency of the cached device, the larger the window of time during which demotion improves performance.

The impact of the demotion depends on the ability of the CPU to overlap the demotion with other computations. If no reads are performed (left side of the graphs of Figure 5), the fence happens just after the demotion order, so demotion cannot be overlapped with other operations. In that case, demotion provides no performance gain.

Slightly more counter-intuitively, the demotion operation also provides no performance gain when reading too much data from the L1 cache (right side of Figure 5). In this case, the runtime of the benchmark becomes dominated by the

reads, and therefore the percentage benefits asymptotically go to 0.

Surprisingly, the impact on performance is highly dependent on the latency of the FPGA. The fast FPGA is most impacted by writes shortly followed by a fence, while the slow FPGA is most impacted by writes further ahead of the fence. The performance difference is explained as follows: when the cache receives the order to make a write visible, it needs to acquire the cache line in exclusive mode, and it needs to read the full cache line prior to updating it. Both operations are impacted by the latency of the FPGA:

- The time to read the cache line is obviously impacted by the latency of the FPGA.
- Acquiring the exclusive right to the cache line is also impacted by the latency of the FPGA. In many modern cache implementations, the cache directory is located on the cached device, instead of being stored in the cache itself. Intel CPUs, for instance, store their directory in DRAM/PMEM. Similarly, the ARM core maintains the status of the cached FPGA memory in the FPGA rather than in its cache. Every cache line status change thus requires accessing the FPGA.

So, the higher the latency of the FPGA, the larger the time needed to read a cache line and to update the cache directory.

Summary. The delayed propagation of writes impacts the performance of applications that use fences or other instructions that enforce memory ordering (atomic operations, etc.). The impact depends on the latency of the cached device but, as CPUs are expected to cache an increasingly wider range of devices, it is likely to impact a wide range of workloads in the future.

5 Potential pitfalls of pre-stores

In the previous sections, we have shown that the lack of sequentiality in CPU evictions and the delayed propagation of write operations can have a high impact on performance. We have seen that the cache behavior can be controlled using pre-stores, leading to performance improvements.

When used correctly, pre-stores improve the management of the cache with little overhead. For instance, in the common case, *cleaning* a cache line simply enqueues a cache line in the write combining buffers of the CPU, which takes on average 1 cycle on our machines.

Inappropriate uses of pre-store. Care must, however, be taken with the use of pre-stores to achieve the desired performance benefits. For instance, Listing 3 presents a slight variation of Listing 1 that constantly rewrites the same cache line instead of writing random elements of an array. In this microbenchmark, *cleaning* the cache line results in unnecessary writebacks to memory – without the pre-store, the data would just be overwritten in the cache. In such an extreme example, pre-stores result in a 75× slowdown – an unsurprising result, equivalent to the ratio between the latency of writing to memory vs. writing to the cache.

Listing 3 Pseudo-code used to illustrate the overhead of pre-storing a frequently rewritten cache line.

```
1  char data[CACHE_LINE_SIZE];
2  while(...) {
3      memset(data, ..., CACHE_LINE_SIZE);
4      prestore(data, CACHE_LINE_SIZE, clean);
5  }
```

Skipping the cache. In Listing 1 we have shown that cleaning is a simple and efficient way to avoid write amplification. We now consider a slight variation of Listing 1 in which Line 5 (the summation) is removed. In this case, skipping the cache is more appropriate because the data is no longer re-read, and therefore there is no reason for it to remain in the cache. Skipping the cache also directs the CPU to write data sequentially, and thereby also eliminates write amplification. Vice versa, in the original version of Listing 1, skipping is 2× slower than cleaning, when using small elements, because skipping causes `elts[idx].field` to be read from memory instead of being read from the cache.

In large code bases, understanding where to add pre-stores by looking at application code can be time-consuming and error-prone. For instance, in the example of Listing 3, the rewriting of the cache line is obvious, but in more realistic code, it may appear in a different function, making it less visible to the developer. To address this problem, we have developed a tool called Dirtbuster. Dirtbuster analyzes the memory access patterns of an application. Based on that, it identifies the specific scenarios and locations where inserting pre-stores or skipping is beneficial. In the case of pre-store, Dirtbuster also suggests which type of pre-store to use.

6 DirtBuster: design and implementation of a tool for pre-stores

In this section, we present the design and implementation of DirtBuster. The goal of DirtBuster is to help developers find the code locations that could benefit from the addition of pre-stores.

6.1 Overview

As seen in Section 4, pre-stores improve the performance of sequential writes and of writes followed by memory ordering constraints. In order to find code sections that match these patterns, DirtBuster relies on dynamic analysis. Figure 6 illustrates the process.

Step 1. The first step of DirtBuster consists of finding the write-heavy code sections of an application. This information is needed to know which functions to patch, and where

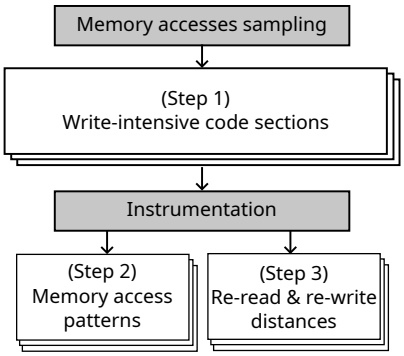

Figure 6. DirtBuster relies both on sampling and binary instrumentation. Sampling allows us to find the write-intensive functions, and binary instrumentation allows us to analyze the access patterns of these functions (sequential writes, writes before a fence, and re-read and re-write distances).

to add pre-stores in the function. To get this information, DirtBuster relies on sampling memory accesses using performance counters.

Step 2. Pre-stores improve performance when the application performs sequential writes or when writes are followed by memory fences. We empirically found that the code surrounding write-intensive regions tends to be spread out in many files and libraries. For instance, atomic instructions, which also impose memory ordering constraints, tend to be called from external libraries (e.g., the atomic instructions of locks are generally called from the `pthread` library). Such external dependencies made statical analysis of the code impractical. DirtBuster thus analyzes the binary of the application and of its libraries.

DirtBuster uses the Intel PIN tool [14] to log all reads and writes performed by the write-intensive functions, and to log all calls to memory fences. The logs are analyzed to check if the code writes data sequentially or if writes are followed by fences.

Step 3. For every memory region that is either written sequentially or written before a fence, DirtBuster helps developers to choose between *demoting* the data, *cleaning* the data or *skipping* the cache. The choice is based on the frequency at which the data is re-read or re-written. For instance, if the data is frequently rewritten, *cleaning* or *skipping* the cache would result in unecessary writes to memory (instead of simply being overwritten in the cache, the data would be pushed to memory every time).

To that end, we compute the re-write and re-read distances of every cache line. The re-write distance is defined as the number of assembly instructions between two consecutive writes to the same cache line. Similarly, the re-read distance is defined as the average number of assembly instructions executed between a read from a cache line and the preceding write to that cache line.

Intended usage. DirtBuster is meant to be executed offline, as an optimization pass before releasing performance-critical applications. As a consequence, we did not focus on minimizing the overhead of the binary instrumentation pass. If an application can only be profiled and tested in performance-critical scenarios, it is possible to skip steps 2-3. In that case, it is up to the developer to understand if the write-intensive functions write data sequentially and if the data is re-used.

Using both sampling and binary instrumentation. The use of both sampling (in step 1) and binary instrumentation (in steps 2 and 3) may seem redundant, but we found it useful in practice.

Sampling memory accesses is useful in step 1 because it allows understanding the behavior of an application with limited overhead (less than 1% in practice). The write-intensive functions found using sampling are thus likely to be the ones that impact performance the most. However, sampling is too imprecise to be used in steps 2-3 to understand the memory access patterns of an application (e.g., sampling one memory access every 10K instructions is too coarse grain to detect sequential strides or to compute re-read or re-write distances accurately).

Conversely, binary instrumentation is too intrusive to be used in step 1: the instrumentation has a large overhead (up to 25× slowdown in practice), and PIN's instrumentation causes cache thrashing, which may cause some cache-friendly functions to wrongly appear as memory-intensive. However, binary instrumentation allows gathering an exact view of all reads and writes performed by an application, which is necessary in steps 2-3 to understand the memory access patterns of the application.

6.2 Implementation

In this section, we present the implementation of DirtBuster, following the conceptual steps of Figure 6.

6.2.1 Step 1: Detecting write-intensive functions. DirtBuster samples memory accesses to find the functions that write data. Finding the functions is useful to know where to add the pre-store instructions.

DirtBuster relies on `perf` to sample the loads and stores performed by an application. DirtBuster gathers the time of all loads and stores, their instruction pointer (IP), and a callchain. The IPs are then grouped by functions to infer the most write-intensive functions. DirtBuster also groups the IPs of the callchains, to infer the most common paths that lead to these functions. Indeed, writes often happen in generic library functions (e.g., `memcpy`), and knowing the callchains that lead to these functions allows patching the relevant application code.

6.2.2 Step 2: Analyzing memory access patterns. In a second step, DirtBuster infers the memory access patterns of the write-intensive functions using binary instrumentation. DirtBuster instruments binaries using Intel PIN [14] to record all reads and writes performed by the write-intensive functions found in the previous step.

Detecting sequential writes. A naive approach to check if a program writes data sequentially is by checking each write operation to determine if it targets the same cache line as the previous write or an adjacent cache line. However, this approach is insufficient for applications that write temporary values in between sequential writes, or for applications that interleave sequential writes to multiple objects.

To avoid such problems, DirtBuster keeps tracks of multiple "sequentiality contexts". A "sequentiality context" is a record of a memory region (range of virtual address) and the location of the last write within that region. When a write is performed, DirtBuster checks if it is adjacent to the last write performed in any "context". If a context is found, its metadata is updated, otherwise a new context is created.

We currently do not impose a limit on the number of contexts tracked by DirtBuster. In practice, we found that the write-intensive functions perform sequntial writes on only a few objects (e.g., a few large arrays).

At the end of the execution, for every write-intensive function, DirtBuster reports the percentage of the writes that happen in sequential contexts and the size of the contexts. For instance, DirtBuster may show that a `functionX` performs 50% of its writes in sequential contexts, and that 80% of the sequential writes are to regions of size 1KB and that the remaining 20% of the sequential writes are to regions of size 16KB.

Detecting memory ordering constraints. To detect memory ordering constraints, DirtBuster computes the minimum number of instructions between the writes performed by the write-intensive functions and the next instruction with fence semantics. Instruction with fence semantics comprise memory fence instructions (e.g., `mfence`, `sfence`, ...) and the atomic instructions that force the CPU to order memory accesses (e.g., `cmpxchg` orders reads and writes).

6.2.3 Step 3: Re-read and re-write distance. DirtBuster computes the re-read and re-write distance of every cache line accessed by the write-intensive functions. To that end, DirtBuster maintains a counter of the number of executed instructions. For every monitored sequential context and for every cache line written before a fence, DirtBuster stores the value of the counter at the latest recorded read and at the latest recorded write. The information is currently stored in a B-Tree.

Whenever a monitored context is re-written, its re-write distance is updated. The only exception is that, to prevent categorizing sequential writes as multiple rewritings of the same context, DirtBuster updates the rewrite distance only when a write breaks a streak of sequential accesses. We compute the re-write distance as the average counter value (average number of instructions) between two subsequent writes to the same cache line. A similar computation is done for the re-read distance.

Guiding developers. When a write intensive function writes data sequentially, or when writes are followed by fences, DirtBuster proposes to use pre-store. DirtBurster reports the name of the function and the line(s) of code that perform writes. From the line(s) of code that perform writes, it is usually obvious to infer which variables are written, and so which variables to pre-store.

If the data is re-written, DirtBuster suggests inserting a *demote* pre-store because it allows making the data visible to the other CPUs before hitting the fence, but keeps the data in the cache to speed up the future re-writes. If the data is just re-read, DirtBuster suggests inserting a *clean* pre-store after writing the data. The *clean* directs the CPU to initiate a write-back of the dirty data, but keeps it in the cache to speed up future re-reads. If the data is not re-read nor re-written, DirtBuster suggests to *skip* the cache using non-temporal stores. If non-temporal stores are hard to implement, a developer can choose to *clean* the cache instead.

The analysis performed by DirtBuster is independent of the machine architecture, but the performance benefits of following Dirtbuster's recommendations in terms of pre-stores depends highly on the architecture's characteristics. For instance, on machine A, with the strong x86 memory model, little gain is to be expected from following the recommendations in terms of writes before a fence (the memory model forces the CPU to make writes visible in order, so writes are rarely kept private in the CPU buffers). Likewise, on machine B, with the cache line and the memory unit having the same size, no benefit is to be expected from the recommendations in terms of sequential writebacks.

7 Evaluation

In this section, we aim at answering the following questions:

- Are pre-stores useful on real applications, and how big is the performance gain?
- Is DirtBuster giving the right recommendations? What is the effort of patching applications?
- Is there any downside in using pre-stores?

7.1 Setup

Machines A and B are described in Section 3. Table 2 presents the applications used in the evaluation. They contain a subset of the Phoronix benchmark suite[1] [32], two key-value stores; a subset of the NAS benchmarks [5], a compilation of HPC

[1]The Phoronix benchmark suite contains 497 test suites, many of which contain multiple applications. For practical reasons, we only executed a subset.

workloads, and X9 [17], a message passing library. Since pre-stores are most useful when writing data, we focus our evaluation on write-intensive applications. In particular,

- Some applications spend less than 10% of their time issuing store instructions (we used `perf` to get this information). Adding pre-stores to these applications would have no effect. We did not instrument these applications further.
- We use DirtBuster to instrument the remaining applications. When DirtBuster recommends adding pre-stores, we patch the application according to DirtBuster's guidance. To further illustrate the usefulness of DirtBuster guidance, we also patch a few applications in which DirtBuster did not find any interesting pattern and show the overheads of pre-stores in such cases.

We focus the evaluation on TensorFlow [1], a machine learning framework, the 9 applications of the NAS benchmark suite and two variants of a key-value stores, one using a CLHT [16] hashmap as index, the other using a Masstree [31] index, and the X9 [17] message passing library.

7.2 Improving sequentiality (on Machine A)

We first evaluate applications in which DirtBuster detects sequential writes.

7.2.1 Machine learning workload. We benchmark TensorFlow, training a CNN. We execute the default ML training workload of the `pts/tensorflow` benchmark of the Phoronix benchmark suite [32]. We use the default input parameters and vary the batch size between 0 and 250. The batch size determines the number of images used simultaneously in an iteration of training; the higher the batch size, the larger the tensors used during training.

Step 1: finding the function to patch. The sampling phase of DirtBuster shows that most writes to memory are performed in a templated function of the Eigen tensor library [19]: `Eigen::TensorEvaluator<...<op>...>::run()`.
The `<op>` template parameter determines the operation performed on the tensor. For instance `Eigen::internal::scalar_sum_op<>` sums two tensors and `Eigen::internal::scalar_product_op<>` performs a scalar product of two tensors. Collectively, all the templated versions of the function account for 50% of the writes to memory when the benchmark is launched with a small batch size (0-50) and for 30% with large batch sizes (50+). DirtBuster indicates that most writes are performed in the loop that calls the inlined `evalPacket` function, which evaluates the operation and writes the result to a tensor (see Listing 4).

Step 2: understanding the memory access patterns. Understanding the code of Eigen library of TensorFlow is beyond the scope of this paper, but we used the binary instrumentation provided by DirtBuster to understand the memory behavior of the unrolled loop:

```
Eigen::TensorEvaluator<...<op>...>::run()
Location: <...>/TensorExecutor.h line 272
Perc. Seq. Writes: 50%
Size: 16.2MB - 10% - re-read inf - re-write inf
...
Size: 240B - 60% - re-read 2 - re-write inf
Pre-store choice: clean
```

The `Eigen::internal::scalar_sum_op<>` function is templated and used to perform tensor operations on tensors of various sizes. Fifty percent of the writes performed by the function occur in sequential contexts. Of these, ten percent involve 16.2MB tensors, while sixty percent involve 240B tensors. For the 16.2MB tensors, the re-read and re-write distances are effectively "infinite," meaning the tensor is neither re-read nor re-written. In contrast, the smaller tensors are re-read almost immediately, with an average of just two instructions between the write and the re-read.

Table 2. Applications used in the evaluation. Some applications are not write-intensive, and would not benefit from pre-stores. For write-intensive applications, we used DirtBuster to detect memory access patterns.

	Write-Intensive	Sequential writes	Writes before fence
pytorch	✗		
numpy	✗		
lzma	✗		
c-ray	✗		
arrayfire	✗		
build-kernel	✗		
build-gcc	✗		
gzip	✗		
go-bench	✗		
rust-prime	✗		
TensorFlow	✓	✓	
X9	✓	✓	✓
Key-Value Stores			
Masstree	✓	✓	✓
CLHT	✓	✓	✓
NAS benchmarks			
UA	✓	✓	
LU	✗		
EP	✗		
IS	✓	✗	
FT	✓	✓	
CG	✗		
BT	✓	✓	
MG	✓	✓	
SP	✓	✓	

Listing 4 Pseudo-code of a section of Eigen library used by TensorFlow to do tensor computations.

```
1  void Eigen::TensorEvaluator<...>::run(...) {
2    ...
3    for (; i <= last_chunk_offset; i += ...) {
4      evaluator.evalPacket(i + 0 * PacketSize);
5      evaluator.evalPacket(i + 1 * PacketSize);
6      evaluator.evalPacket(i + 2 * PacketSize);
7      evaluator.evalPacket(i + 3 * PacketSize);
8      prestore(&evaluator.data()[i], 64, clean);
9    }
10   ...
11 }
```

Step 3: choosing the correct type of pre-store. Because the function is templated, the same code is used to operate on the 16.2MB tensors as on the 240B tensors. Since the re-read distance for the 240B tensor is very small, DirtBuster suggests using a *clean* pre-store. This choice would unlikely be the default choice of a developer, because nothing in the code suggests that the data is reused, and the manually unrolled loop can easily be modified to use non-temporal instructions. Without DirtBuster a developer would likely choose to *skip* the cache.

Performance. To check the correctness of the recommendations of DirtBuster, we evaluate the performance improvements of both *cleaning* and *skipping*. *Cleaning* required adding a single pre-store line, while *skipping* required modifying the `evalPacket` function to use non-temporal instructions. The performance improvements on Machine A are presented in Figure 7. The performance improvement depends on the batch size used for the training. With small batch sizes, the performance improvement of *cleaning* is up to 47% (batch size 1), dropping to 20% with larger batch sizes. As DirtBuster suggested, *cleaning* the cache is the right optimization, and using non-temporal writes (*skipping* the cache) reduces performance by 20%.

Such large performance variations after patching a single function of TensorFlow may seem surprising, but, as mentioned earlier, the function represents half of the writes to memory for small batch sizes and a third for larger batch sizes. Without pre-storing, TensorFlow is memory-bound, limited by the speed at which the CPU evicts data. As illustrated in Figure 8, without *cleaning*, write amplification is 3.7×, while with *cleaning*, it drops to 2.7×, explaining the performance gains.

In these experiments, we only patched a single function of TensorFlow, which explains why pre-storing does not fully eliminate write amplification. DirtBuster detects other write-intensive functions, but indicates that they do not write data sequentially, and therefore we did not modify them to add pre-stores. Trying to do so, unsurprisingly, had no effect on performance. While pre-stores have no noticeable overhead, in this case they have no effect on the sequentiality of evictions and therefore offer no performance improvement.

Using non-temporal stores results in a performance loss because the newly written values depend on previously written values. The `evalPacket` function starts by loading a previously written packet, computes a new value and then stores the new value (i.e., a pattern similar to `a[x] = f(a[x-4*PacketSize])`). Profiling shows that `skipping` the cache doubles the time spent loading the value of the previously written packet.

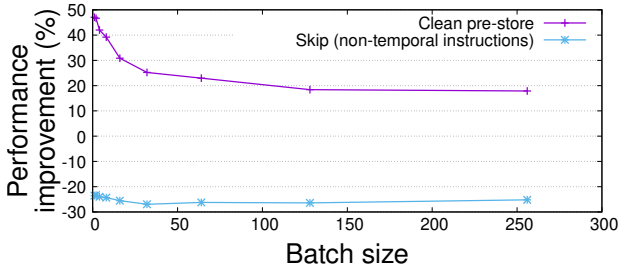

Figure 7. Machine A. TensorFlow. Performance improvement brought by pre-storing data. As advised by DirtBuster, cleaning the cache improves performance, while skipping the cache decreases performance.

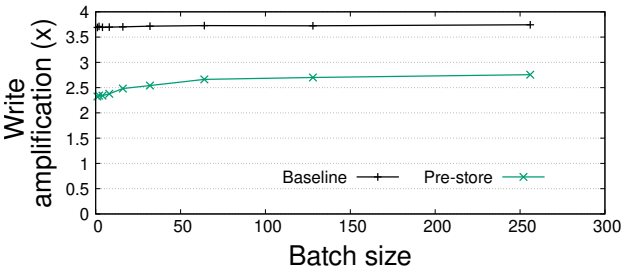

Figure 8. Machine A. TensorFlow. Adding a cleaning pre-store significantly reduces write amplification.

7.2.2 HPC workloads. We evaluate applications from the NAS benchmark suite. From all applications contained in the suite, only MG, FT, SP, UA and BT spend more than 10% of their time writing data.

Analysis of MG. MG performs a multi-grid method on a sequence of meshes and is implemented as a succession of matrix multiplications. MG allocates 3 matrices, U, V and R. DirtBuster detects that the `psinv` function writes the `U` matrix sequentially and that the `resid` function writes the R matrix sequentially. For the `resid` function, the output of DirtBuster is as follows:

```
Location: <...>/mg.f90 line 544
Perc. Seq. Writes: 100%
```

```
Size: 2.1MB- 100% - re-read 23.8K - re-write inf
Pre-store choice: clean
```

For the psinv function, the output looks as follows:

```
Location: <...>/mg.f90 line 614
Perc. Seq. Writes: 100%
Size: 2.1MB - 100% - re-read inf - re-write inf
Pre-store choice: skip
```

The data is written in chunks of 2.1MB. DirtBuster suggests *cleaning* the cache in the `resid` function because the data is re-read, and *skipping* the cache for the `psinv` function because the data is not re-read not re-rewritten ("infinite" number of instructions between two accesses to the same cache line). Unfortunately, Fortran offers no standard way to use non-temporal stores, so we chose the next best option, i.e., to *clean* the cache in both functions. Listing 5 presents the 1-line change to the `psinv` function. A similar change is made to the `resid` function. None of the authors had any prior knowledge of Fortran before changing the code of the NAS benchmarks, but DirtBuster helped pinpoint the exact matrix and code location that could benefit from pre-storing, and adding the pre-store instructions took less than an afternoon of work.

Listing 5 Change made to the `psinv` function of MG.

```
1  subroutine psinv( r,u,n1,n2,n3,c,k)
2  !$omp parallel do ...
3  do i3=2,n3-1
4    do i2=2,n2-1
5      do i1=2,n1-1
6        u(i1,i2,i3) = u(i1,i2,i3)
7             + c(0) * r(i1,i2,i3) + ...
8      enddo
9      call prestore(loc(u(2,i2,i3)), ..., clean)
10   enddo
11 enddo
```

Analysis of FT, SP, UA and BT. FT performs a Fast Fourier Transform. DirtBuster indicates that the `cffts1` function sequentially transfers results from a matrix `Y1` to a matrix `XOUT`. SP is a Scalar Penta-diagonal solver. DirtBuster detects that SP allocates dozens of matrices, but that a single matrix (`RHS`) accounts for most of the writes. The matrix is mostly written in the `compute_rhs` function and is rarely reused. Similarly to the change done in MG, we chose to *clean* the matrices after writing them. Similarly, UA and BT write matrices that we also chose to *clean*.

Performance. Figure 9 summarizes the performance of the NAS benchmarks with pre-storing on Machine A. Pre-storing is up to 40% faster.

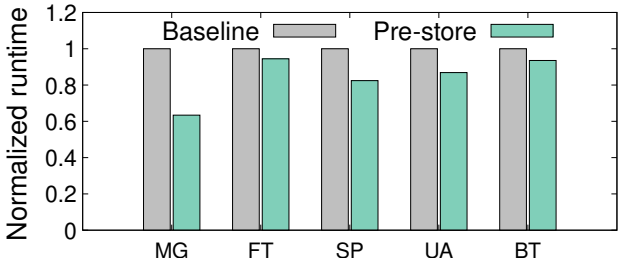

Figure 9. Machine A. Normalized runtime of the NAS benchmarks. Lower is better.

7.2.3 Key-Value stores. In this section, we evaluate the performance of the CLHT and Masstree key-value stores, running a YCSB workload. We configure the key-value stores with 100 million keys and evaluate value sizes ranging from 64B to 4KB. We inject load using 10 threads, the configuration that provides the highest throughput.

Analysis. When executing YCSB, DirtBuster indicates that read-only or read-mostly workloads (YCSB B-D) do not benefit from pre-storing data. However, on YCSB A (50% GET, 50% PUT), DirtBuster suggests *skipping* the cache when executing a PUT query. DirtBuster indicates that the values inserted in the key-value store are rarely reused, that they are written sequentially, and that the writes are followed by a fence.

Performance. We compare the performance of the unmodified code (baseline) against versions using pre-stores. In the first version, we used non-temporal stores to craft the values of the PUT function (suggestion of DirtBuster). In the second version, we clean the cache after crafting the value. Listing 6 presents the version of the PUT function of CLHT when cleaning the *cache*. *Skipping* the cache required rewriting the `craftValue` function).

Listing 6 Pseudo-code of PUT function of CLHT.

```
1 void ycsb_put(...) {
2     void *value = craftValue(...);
3     prestore(value, size, clean);
4     clht_put(..., value);
5 }
```

Figures 10 and 11 summarize the performance of CLHT and Masstree on Machine A, running YCSB A. Skipping the cache is up to 2.9× faster than the baseline for CLHT and 2.5× faster for Masstree.

As advised by DirtBuster, *skipping* the cache offers the best performance, but it also requires significant changes to the `craftValue` function. *Cleaning* only requires adding a single line of code and is still up to 2.3× faster than the baseline for CLHT and up to 1.9× faster for Masstree. In

complex code bases, it is thus possible to simply *clean* the cache to achieve significant performance gains.

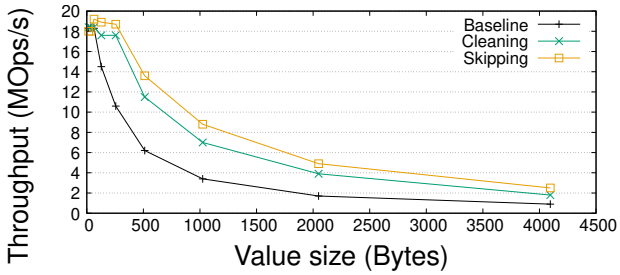

Figure 10. Machine A. CLHT performance, in requests per second. Higher is better.

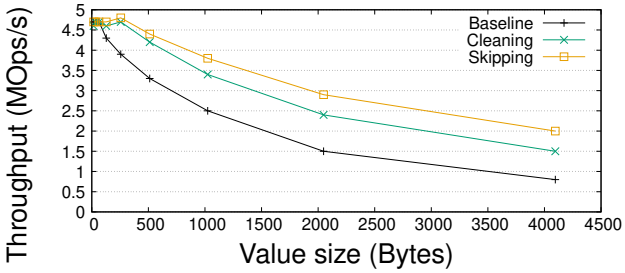

Figure 11. Machine A. Masstree performance, in requests per second. Higher is better.

Profiling shows that skipping and cleaning the cache reduce write amplification. When running YCSB with values larger than 256B, for the baseline every 64B evicted from the cache results in an average of 248B written to persistent memory (3.8× write amplification, see Figure 12). Skipping and cleaning both eliminate write amplification when YCSB is configured to use large values. Skipping outperforms cleaning because the crafted values no longer pollute the cache.

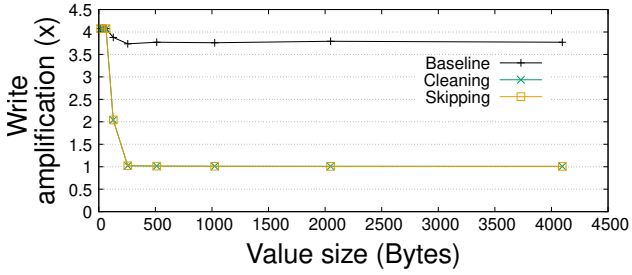

Figure 12. Machine A. Write amplification of CLHT executing YCSB A (lower is better).

As expected, the reduction in write amplification is the largest when the value size matches or exceeds the cache line size of persistent memory (256B). Pre-storing, however, improves performance as soon as the value size exceeds the cache line size of the CPU (64B). For instance, with 128B values, pre-storing halves the write amplification and improves performance by 21.4% for CLHT and 7.0% for Masstree.

7.3 Improving latency (on Machine B)

In this section we evaluate applications in which DirtBuster detects writes followed by memory ordering constraints.

7.3.1 Key-value stores. We run the YCSB workload presented in the previous section on machine B.

On machine A, pre-storing is useful because it increases the sequentiality of evictions. On machine B, the FPGA interleaves memory requests to multiple concurrent memory controllers, so the machine does not benefit from the increase in sequentiality. Pre-storing is, however, still useful because writes are followed by fences and atomic instructions. To measure the performance of *cleaning*, we use the same patch as the one used for Machine A (see Listing 6). We also tried to port the code used to *skip* the cache on machine A, but could not implement a working version for machine B (non-temporal code is architecture-specific, and Arm CPUs do not offer standard libraries to implement non-temporal operations). As a consequence, we only present *cleaning* results in this section.

Figures 13 and 14 present the performance with 1KB values. Pre-storing is 52% faster on CLHT and 25% faster for Masstree. Pre-storing is most useful when machine B is configured to use the fast FPGA because the memory ordering instructions happen soon after writing data.

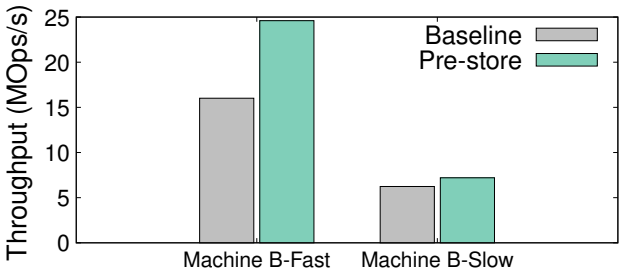

Figure 13. Performance of CLHT on Machine B-fast (FPGA configured with a low latency) and Machine-B-slow (high-latency FPGA).

When inserting an object, CLHT computes the hash of the object and then locks the bucket it belongs to. The atomic operations used in the lock have a fence semantics and force the CPU to make the crafted value visible to all the cores in the machine. Profiling shows that pre-storing allows making the value visible before reaching the atomic operation, reducing the time spent in the atomic instructions of the lock by 74%.

Similarly, to ensure correctness, Masstree uses version numbers for all of its objects. When reading or updating an

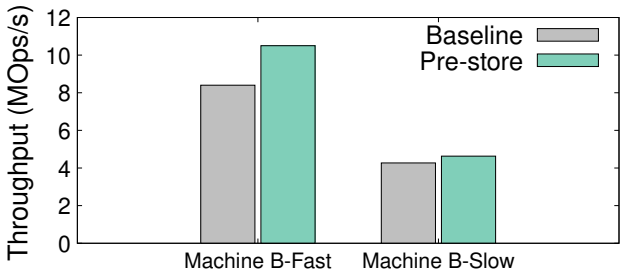

Figure 14. Performance of Masstree on Machine-B-fast and -slow.

object, Masstree checks the version number of the object and then re-checks the version number after manipulating the object, to detect possible concurrent changes. To enforce the correct order of the operations, Masstree uses fences. Listing 7 presents the logic of the code for the insertion of a new value in the KV. The fences are mandatory for correctness, but they may cause the CPU to stall if the crafted value has not been made visible to all the cores of the machine. Profiling shows that pre-storing the values halves the time spent in the first fence of `masstree::put`.

Listing 7 Pseudo-code of the actions performed while inserting a new value in Masstree. The fences ensure correctness, but impact the pipeline of the CPU.

```
1  void *masstree::put(...) {
2      while(...) { // tree traversal
3          v = node->readVersion();
4          if(isLocked(v))
5              goto restart;
6          fence();
7          ... // read or modify the node
8          fence();
9          if(node->versionChanged(v))
10             goto restart;
11     }
12 }
```

7.3.2 Message passing workload. We benchmark the X9 message passing library. We measure the latency of sending messages from a writer thread to a reader thread (see Listing 8). DirtBuster detects that the crafting of a message, in function `fill_msg` is followed by a compare-and-swap instruction in the `x9_write_to_inbox` function. DirtBuster also detects that the benchmark rewrites the same message structures multiple times: X9 reuses the message structures to avoid the overheads of allocations on every message exchange. DirtBuster recommends to *demote* the messages after writing them.

We added a *demote* pre-store after `fill_msg`. The pre-store reduces the latency of sending a message by 62% on machine B-fast, and 40% on machine B-slow. Profiling shows that the pre-store reduces the time spent in the compare-and-swap. Without pre-store, the message is kept in private CPU buffers and is only made visible "at the last minute", when executing the compare-and-swap. With pre-stores, the message is sent to the L2 cache asynchronously, before reaching the compare-and-swap instruction.

Listing 8 X9, pre-storing before sending a message.

```
1  void* producer_fn(...) {
2   fill_msg(&m[...]);
3   prestore(m[...], sizeof(msg), demote);
4   x9_write_to_inbox(inbox, sizeof(msg), m[...]);
5  }
```

7.4 Overhead of pre-stores

In this section, we analyse the overhead of pre-stores, when they are not needed. We distinguish pre-stores suggested by DirtBuster, on architectures that do not benefit from it, and pre-stores that we tried to add manually, without DirtBuster guidance.

7.4.1 Pre-stores suggested by DirtBuster. The NAS applications and TensorFlow only use a fraction of the available bandwidth of Machine B, and they do not use fences. As a consequence, pre-stores do not provide any benefit on machine B. We still evaluated the performance of the patched applications, cleaning data after writing the tensors and the matrices. Unsurprisingly, adding pre-stores does not have a positive impact on performance, but the maximum overhead was limited to 0.3%. In general, we could not find applications for which adding pre-stores at the locations recommended by DirtBuster added any significant overhead.

7.4.2 Incorrect manual use of pre-stores. For completeness, we tried to patch some functions and applications in which DirtBuster does not recommend using pre-stores.

FT from the NAS benchmark suite. DirtBuster suggested adding pre-stores to the `cffts1` function of FT, which resulted in performance gains on machine A (see Section 7.2.2). When profiling the application with `linux perf`, we noticed that another function, called `fftz2`, is also write-intensive. While a manual reading of the code suggests that it is sequentially writing data, it is not immediately apparent that the written data is small enough to fit in the cache and is frequently re-read and re-written. *Cleaning* the cache in that function resulted in a 3× slowdown. DirtBuster was able to detect both the length of the sequential writes and the distance between rewrites, and did not suggest using a pre-store.

IS from the NAS benchmark suite. While manually profiling IS using `perf`, we observed that a single function, `rank`,

is responsible for the majority of writes. Similar to the previous case, this function contains a small loop that might initially seem like a suitable candidate for pre-stores. However, the function actually writes small amounts of data in a seemingly random pattern. In this case, adding a pre-store has no effect (no performance gain, no overhead) because the written data is neither re-read nor re-written. DirtBuster detects the lack of sequentiality and does not suggest using a pre-store.

Conclusion. The examples above demonstrate that using DirtBuster is a valuable approach when optimizing memory write operations in applications. DirtBuster not only identifies scenarios where pre-stores (or skips) can enhance performance, but also refrains from recommending their use where they would not result in any gains or could even produce slowdowns.

8 Related Work

Directing the cache. The idea of controlling hardware caches in software dates back from the 80s [36]. Most work since then has focused on helping the caches to prefetch data in order to hide the latency of memory, either by improving prediction capabilities in hardware [21, 35] or by directing caches with software prefetch instructions [21, 27, 30, 33–35, 37, 46]. More recently, the advent of persistent memory has brought back the necessity of cleaning cache lines to offer persistence guarantees. Most published work has focused on detecting incorrectly placed cleaning instructions to find persistence bugs [18, 29]. To hide the high latency of writing data to PMEM [38, 51], Shin et al. [42] and Ribbon [47] have proposed to evict dirty data to persistent memory in the background. Xu et al. [49] have proposed to cache written data with higher priority to avoid unnecessary writebacks. In DirtBuster, we propose the more general concept of a pre-store and show its usefulness to improve sequentiality and to reduce latency on weak memory architectures.

Software approaches to influence caching. Multiple strategies have been proposed for maximizing the efficiency of CPU caches in software. Khan et al. [25] have proposed to partition the cache to avoid conflicts between read-mostly and write-mostly data. In general, the careful placement of pages in virtual and physical memory [6, 24, 28] has been used to reduce conflicts in the cache. Page coloring techniques partition the cache to avoid cache thrashing between users or applications [7, 50]. Scheduling techniques ensure that critical applications are co-scheduled with applications that do not thrash their CPU caches [4, 44, 52]. These techniques are orthogonal to the ones explored in this paper.

Sequentiality. The importance of sequentially accessing cached devices has been mentioned in previous work. Most work have focused on designing data structures that can be read and written in long sequential strides [23, 48]. While the code accesses data sequentially, no guarantee is provided at the hardware level. DirtBuster could be used to enforce that the CPU evicts data in sequential order.

Cleaning the cache. In this paper, we use cleaning instructions to improve performance. Cleaning the cache has also been used to reduce the likelihood of timing attacks [22].

The term "pre-store" in the literature. The term "pre-store" was used in the related work to refer to different concepts. Chen et al. [8] propose to add a non-cache coherent buffer on top of a cache. Data is "pre-loaded" in the buffer but, because the buffer is non cache-coherent, it cannot contain entries that are later written. What the paper defines as a "pre-store" is an instruction that tells the buffer to not "pre-load" a given memory address in the buffer. Kim et al. [26] focus on the DRAM cache of SSDs. They propose to automatically evict dirty data from the DRAM cache to storage in the background, when the storage is idle (the background eviction scheme is referred to as "pre-storing"). Doing a similar strategy for the CPU cache would require hardware changes because only the CPU knows when its memory controller is idle. In our context, a "cache pre-store" is an assembly instruction that directs the CPU to move data down the memory hierarchy.

9 Conclusion

We have introduced the concept of pre-stores for moving data asynchronously down the memory hierarchy – the converse of pre-fetches that move data up in the memory hierarchy. We have demonstrated the benefits of pre-stores on two types of architectures: one in which the CPU cache line size is different from the write unit of the underlying memory and one in which updating the cache exhibits long latencies.

To assist the developer in inserting pre-stores in judicious program locations and in avoiding potential pitfalls in doing so, we have presented the Dirtbuster tool, a dynamic analysis tool that uses memory access sampling and instrumentation to discover code patterns suitable for insertion of pre-stores.

We have used Dirtbuster on a large number of benchmarks. We have demonstrated performance improvements of up to 2.3× on some benchmarks, without incurring performance regression on any of them.

Acknowledgements. We would like to thank our shepherd, Horst Schirmeier, and the anonymous reviewers for all their helpful comments and suggestions. This work was supported in part by the Australian Research Council Grant DP210101984. We thank the Enzian team at ETH Zurich for access to the Enzian machines. We thank Shuaiwen Song and the members of the FSA lab for their guidance and feedback throughout the project.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Inc Advanced Micro Devices. Ai acceleration with amd radeon. https://www.amd.com/en/products/graphics/radeon-ai.html, 2024.

[3] Arm. L210 Cache Controller Technical Reference Manual. https://developer.arm.com/documentation/ddi0284/g/Babeegje, 2023.

[4] Reza Azimi, David K Tam, Livio Soares, and Michael Stumm. Enhancing operating system support for multicore processors by using hardware performance monitoring. *ACM SIGOPS Operating Systems Review*, 43(2):56–65, 2009.

[5] David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow. The nas parallel benchmarks 2.0. Technical report, Technical Report NAS-95-020, NASA Ames Research Center, 1995.

[6] Brian N Bershad, Dennis Lee, Theodore H Romer, and J Bradley Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 158–170, 1994.

[7] Edouard Bugnion, Jennifer M Anderson, Todd C Mowry, Mendel Rosenblum, and Monica S Lam. Compiler-directed page coloring for multiprocessors. *ACM SIGPLAN Notices*, 31(9):244–255, 1996.

[8] William Y Chen, Roger A Bringmann, Scott A Mahlke, Richard E Hank, and James E Sicolo. An efficient architecture for loop based data preloading. *ACM SIGMICRO Newsletter*, 23(1-2):92–101, 1992.

[9] David Cock, Abishek Ramdas, Daniel Schwyn, Michael Giardino, Adam Turowski, Zhenhao He, Nora Hossle, Dario Korolija, Melissa Licciardello, Kristina Martsenko, et al. Enzian: an open, general, cpu/fpga platform for systems software research. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 434–451, 2022.

[10] Intel Corporation. Discover advanced memory with intel® optane™ pmem. https://www.intel.com/content/www/us/en/products/details/memory-storage/optane-dc-persistent-memory.html, 2024.

[11] Intel Corporation. From intel® optane™ persistent memory to cxl. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory-to-cxl-attached-memory.html, 2024.

[12] Intel Corporation. Intel® accelerator engines for demanding workloads help enhance roi. https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/overview.html#:~:text=Intel%C2%AE%20Accelerator%20Engines%20are,cloud%2C%20and%20at%20the%20edge., 2024.

[13] Intel Corporation. Intel® xeon® cpu max series. https://www.intel.com/content/www/us/en/products/details/processors/xeon/max-series.html, 2024.

[14] Intel Corporation. Pin - a dynamic binary instrumentation tool. https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html, 2024.

[15] Intel Corporation. A utility for configuring and managing intel® optane™ persistent memory modules (pmem). https://github.com/intel/ipmctl, 2024.

[16] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. *ACM SIGARCH Computer Architecture News*, 43(1):631–644, 2015.

[17] Diogo Flores. X9 - High performance message passing library. https://github.com/df308/x9, 2023.

[18] João Gonçalves, Miguel Matos, and Rodrigo Rodrigues. Mumak: Efficient and black-box bug detection for persistent memory. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 734–750, 2023.

[19] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[20] Apple Inc. Apple introduces m4 chip. https://www.apple.com/au/newsroom/2024/05/apple-introduces-m4-chip/, 2024.

[21] Yasuo Ishii, Mary Inaba, and Kei Hiraki. Access map pattern matching for data cache prefetch. In *Proceedings of the 23rd international conference on Supercomputing*, pages 499–500, 2009.

[22] Reiley Jeyapaul and Aviral Shrivastava. Smart cache cleaning: Energy efficient vulnerability reduction in embedded processors. In *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*, pages 105–114, 2011.

[23] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H Noh, and Young-ri Choi. {SLM-DB}:{Single-Level}{Key-Value} store with persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 191–205, 2019.

[24] Richard E Kessler and Mark D Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems (TOCS)*, 10(4):338–359, 1992.

[25] Samira Khan, Alaa R Alameldeen, Chris Wilkerson, Onur Mutluy, and Daniel A Jimenezz. Improving cache performance using read-write partitioning. In *2014 IEEE 20th international symposium on high performance computer architecture (HPCA)*, pages 452–463. IEEE, 2014.

[26] Jin-Young Kim, Tae-Hee You, Hyeokjun Seo, Sungroh Yoon, Jean-Luc Gaudiot, and Eui-Young Chung. An effective pre-store/pre-load method exploiting intra-request idle time of nand flash-based storage devices. *Microprocessors and Microsystems*, 50:222–236, 2017.

[27] Rakesh Krishnaiyer, Emre Kultursay, Pankaj Chawla, Serguei Preis, Anatoly Zvezdin, and Hideki Saito. Compiler-based data prefetching and streaming non-temporal store generation for the intel (r) xeon phi (tm) coprocessor. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pages 1575–1586. IEEE, 2013.

[28] Baptiste Lepers and Willy Zwaenepoel. Johnny cache: the end of DRAM cache conflicts (in tiered main memory systems). In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 519–534, Boston, MA, July 2023. USENIX Association.

[29] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. Pmtest: A fast and flexible testing framework for persistent memory programs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 411–425, 2019.

[30] Chi-Keung Luk and Todd C Mowry. Cooperative prefetching: Compiler and hardware support for effective instruction prefetching in modern processors. In *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 182–193. IEEE, 1998.

[31] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 183–196, 2012.

[32] Phoronix Media. Open-Source, Automated Benchmarking. https://www.phoronix-test-suite.com/, 2023.

[33] Sparsh Mittal. A survey of recent prefetching techniques for processor caches. *ACM Computing Surveys (CSUR)*, 49(2):1–35, 2016.

[34] Todd C Mowry. Tolerating latency in multiprocessors through compiler-inserted prefetching. *ACM Transactions on Computer Systems (TOCS)*, 16(1):55–92, 1998.

[35] Kyle J Nesbit and James E Smith. Data cache prefetching using a global history buffer. In *10th International Symposium on High Performance Computer Architecture (HPCA'04)*, pages 96–96. IEEE, 2004.

[36] Susan Owicki and Anant Agarwal. Evaluating the performance of software cache coherence. *ACM SIGARCH Computer Architecture News*, 17(2):230–242, 1989.

[37] R Hugo Patterson, Garth A Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 79–95, 1995.

[38] Azalea Raad, Luc Maranget, and Viktor Vafeiadis. Extending intel-x86 consistency and persistency: Formalising the semantics of intel-x86 memory types and non-temporal stores. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–31, 2022.

[39] Samsung. Samsung electronics introduces industry's first 512gb cxl memory module. https://news.samsung.com/global/samsung-electronics-introduces-industrys-first-512gb-cxl-memory-module, 2024.

[40] Deeksha Satish and S Manimala. A study on cache replacement policies in coherent chip multiprocessor systems (cmps). *International Journal For Technological Research In Engineering, Volume 5*, 2018.

[41] Debendra Das Sharma, Robert Blankenship, and Daniel S Berger. An introduction to the compute express link (cxl) interconnect. *arXiv preprint arXiv:2306.11227*, 2023.

[42] Seunghee Shin, James Tuck, and Yan Solihin. Hiding the long latency of persist barriers using speculative execution. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 175–186, 2017.

[43] Ben Simner, Shaked Flur, Christopher Pulte, Alasdair Armstrong, Jean Pichon-Pharabod, Luc Maranget, and Peter Sewell. Armv8-a system semantics: instruction fetch in relaxed architectures. In *Programming Languages and Systems: 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings 29*, pages 626–655. Springer International Publishing, 2020.

[44] David Tam, Reza Azimi, and Michael Stumm. Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors. *ACM SIGOPS Operating Systems Review*, 41(3):47–58, 2007.

[45] Henry Wong. Intel Ivy Bridge Cache Replacement Policy. https://blog.stuffedcow.net/2013/01/ivb-cache-replacement/, 2013.

[46] Carole-Jean Wu, Aamer Jaleel, Margaret Martonosi, Simon C Steely Jr, and Joel Emer. Pacman: prefetch-aware cache management for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 442–453, 2011.

[47] Kai Wu, Ivy Peng, Jie Ren, and Dong Li. Ribbon: High performance cache line flushing for persistent memory. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 427–439, 2020.

[48] Kan Wu, Kaiwei Tu, Yuvraj Patel, Rathijit Sen, Kwanghyun Park, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. {NyxCache}: Flexible and efficient multi-tenant persistent memory caching. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 1–16, 2022.

[49] Yuanchao Xu, Yuanyuan Xu, Min Tang, Liangliang Zhang, and Yazhu Lan. Asymmetry & locality-aware cache bypass and flush for nvm-based unified persistent memory. In *2019 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*, pages 168–175. IEEE, 2019.

[50] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 89–102, 2009.

[51] Yiying Zhang and Steven Swanson. A study of application performance with non-volatile main memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10. IEEE, 2015.

[52] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. *ACM Sigplan Notices*, 45(3):129–142, 2010.