WARDen: Specializing Cache Coherence for High-Level Parallel Languages

Michael Wilkins
Northwestern University, USA

Sam Westrick
Carnegie Mellon University, USA

Vijay Kandiah
Northwestern University, USA

Alex Bernat
Northwestern University, USA

Brian Suchy*
Northwestern University, USA

Enrico Armenio Deiana
Northwestern University, USA

Simone Campanoni
Northwestern University, USA

Umut A. Acar
Carnegie Mellon University, USA

Peter Dinda
Northwestern University, USA

Nikos Hardavellas
Northwestern University, USA

Abstract

High-level parallel languages (HLPLs) make it easier to write correct parallel programs. Disciplined memory usage in these languages enables new optimizations for hardware bottlenecks, such as cache coherence. In this work, we show how to reduce the costs of cache coherence by integrating the hardware coherence protocol directly with the programming language; no programmer effort or static analysis is required.

We identify a new low-level memory property, WARD (WAW Apathy and RAW Dependence-freedom), *by construction* in HLPL programs. We design a new coherence protocol, WARDen, to selectively disable coherence using WARD.

We evaluate WARDen with a widely-used HLPL benchmark suite on both current and future x64 machine structures. WARDen both accelerates the benchmarks (by an average of 1.46x) and reduces energy (by 23%) by eliminating unnecessary data movement and coherency messages.

CCS Concepts: • **Computer systems organization** → **Multicore architectures**; • **Computing methodologies** → *Parallel programming languages.*

Keywords: cache coherence, disentanglement

ACM Reference Format:
Michael Wilkins, Sam Westrick, Vijay Kandiah, Alex Bernat, Brian Suchy, Enrico Armenio Deiana, Simone Campanoni, Umut A. Acar, Peter Dinda, and Nikos Hardavellas. 2023. WARDen: Specializing Cache Coherence for High-Level Parallel Languages. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization (CGO '23), February 25 – March 1, 2023, Montréal, QC, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3579990.3580013

*Now at Google

1 Introduction

High-level parallel languages (HLPLs) are an increasingly popular approach to programming shared-memory multiprocessors. HLPLs are memory-managed languages that make parallel programming simpler and safer. We focus on HLPLs that implement the fork-join programming idiom popularized in lower-level contexts such as OpenMP [65]. Example HLPLs include various dialects of Parallel ML [2, 34, 81, 90].

HLPLs are growing in popularity across academia, industry, and education. Academia and industry are actively developing and improving these languages [2, 79, 80, 90]. Also, HLPLs are part of the curricula of top undergraduate programs. For example, Carnegie Mellon University uses HLPLs to teach the fundamentals of data structures and parallel algorithms. Given HLPLs' expanding usage, there is an exigency to accelerate these programs. In this paper, we present a novel hardware-software co-design that leverages a unique property of HLPLs to improve the memory subsystem.

A major bottleneck in the memory system of multiprocessors is cache coherence. Hardware cache coherence protocols manage data within the private and shared caches to implement the shared memory abstraction. Coherence is known to amount to $\approx$ 50% of on-chip interconnect traffic [14, 26, 29, 76]. Future systems, which are projected to burgeon through higher core counts, multi-socket systems [3, 13, 27, 42, 44, 46, 49, 62, 83] or disaggregation [15, 30, 45, 52, 55, 58, 60, 68] will further increase the cost of coherence.

Previous works improved the coherence protocol's performance for select non-HLPL programs [12, 21, 74, 75]. These approaches targeted programs with the data-race freedom

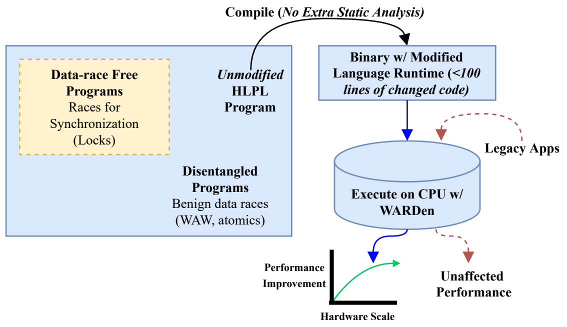

Compile (No Extra Static Analysis)
Data-race Free Programs
Races for Synchronization (Locks)
Unmodified HLPL Program
Disentangled Programs
Benign data races (WAW, atomics)
Binary w/ Modified Language Runtime (<100 lines of changed code)
Legacy Apps
Execute on CPU w/ WARDen
Performance Improvement
Hardware Scale
Unaffected Performance

Figure 1. WARDen accelerates a broad set of applications without user annotation or static analysis while maintaining performance for legacy applications. Acceleration increases with hardware scale.

(DRF) property [1]. DRF is program-specific, so developers must ensure that their program maintains the property and then correctly announce it via manual annotation. Manual annotation is a substantial burden in any case, and in HLPLs, it would defeat the purpose of "high-level" abstractions that insulate developers from their program's memory behavior.

Instead, we target an HLPL property called disentanglement (§2.1) [90]. Disentanglement is broader than DRF because it allows for "benign" races, such as select write-after-write (WAW) dependencies. Intuitively, WAWs may be benign because the algorithm does not care which value persists, or that the written values are the same. Disentanglement is particularly advantageous because it can be embedded in the implementation of an HLPL without user annotation or static analysis. Therefore, HLPL programs can benefit from our improvements *with zero programmer effort*.

We observe that the disentanglement property enables new architectural optimizations. To take advantage, we introduce a low-level memory property, "WARD", and unify it with disentanglement. We first define the *WARD* (Write-after-write Apathy and Read-after-write Dependence-freedom) property, as well as a WARD region, the set of addresses and a period of time over which the WARD property holds (§3). Intuitively, all concurrent updates in a WARD region can be simply reconciled in an unordered manner when the region ends. As a consequence, hardware coherence can be disabled for WARD regions. Also, WARD regions can be identified automatically by the HLPL runtime with zero compile-time or run-time overhead and no programmer involvement (§4).

To utilize WARD regions, we introduce the *WARDen* protocol, a small extension of directory-based cache coherence protocols. WARDen (§5) augments the commonplace MESI protocol [63] with a WARD state, which indicates that coherence should effectively be disabled. The WARDen protocol protects accesses within WARD regions from coherence-driven interruptions (i.e., invalidation and downgrade requests). Our approach maintains the entire functionality of MESI, ensuring that legacy applications run unencumbered.

We implement WARDen within a validated Sniper [16, 17] simulation and extend a Parallel ML language implementation, MPL [61, 90], to signal WARD regions to the hardware.

We evaluate the integrated WARDen system using a variety of benchmarks that encompass the common use-cases of HLPLs [78]. WARDen is able to obtain *significant* performance and energy improvements. We explore the benefits and costs on a wider space of hardware than is currently available, including disaggregated systems. Figure 1 visualizes the overarching story of our work.

We summarize our contributions as follows.

- We define WARD, a property for parallel programs that precludes the need for cache coherence (§3).
- We introduce techniques for identifying WARD automatically in high-level parallel programs and implement them in the MPL runtime (§4).
- We propose WARDen, a cache coherence protocol that allows applications to disable coherence in accordance with the WARD property (§5).
- We develop a simulation-based prototype of WARDen (§6). We find it achieves a 1.46x average speedup while reducing energy usage by 23%. We consider how the benefits of WARDen grow with future hardware (§7).

2 Background

Our work juxtaposes concepts from modern programming languages and computer architecture.

2.1 Disentanglement

Fork-Join: The HLPLs we focus on implement nested fork-join parallelism, which relieves the programmer from manually managing parallelism. They instead use high-level constructs such as parallel-for loops. This approach relies on a thread scheduler (e.g., work-stealing) to create and schedule parallel threads. It allows for fine-grained parallelism by forking and joining many light-weight dynamic threads.

Spawn Trees: During execution, a fork-join program consists of a dynamic tree of light-weight threads called the *spawn tree*. Initially, there is a single root thread. At any moment, any leaf (i.e., a thread with no children) may *fork*, which suspends the thread and spawns two or more new child threads to run in parallel. When all children of a thread have completed execution, they may *join*, which removes the completed children from the tree and resumes the parent as a leaf. In this way, all internal (non-leaf) threads are suspended, and all computation is performed by leaf threads. Two threads are *concurrent* if neither is an ancestor of the other. That is, concurrent threads are siblings, cousins, etc.

Disentanglement: To avoid race conditions, fork-join parallel programs use memory in a disciplined manner. There are a number of approaches to this end, including race-detection [7, 19, 31, 33, 35, 59, 71, 72, 88], type and effect systems [12, 48], programming techniques for determinism [9, 53, 54], as well as determinism-by-default with purely functional programming [2, 4, 8, 10, 34, 37–39, 56, 69, 70, 82, 90].

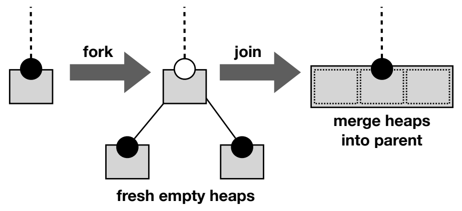


Figure 2. Heap hierarchy at forks and joins [2]. Black circles are active leaf threads, and white circles are suspended (internal) threads. Gray rectangles are heaps.

Recent work identifies one discipline called *disentanglement*, which (informally) means that concurrent threads are oblivious to each other's allocations [2, 37, 70, 89, 90]. Disentanglement allows communication between concurrent threads only through memory allocated by common ancestors.

MPL and the Heap Hierarchy: Disentanglement can be exploited for improved efficiency and scalability, especially for automatic memory management and parallel GC, as demonstrated by the MPL ("maple") compiler [2, 61, 90] for Parallel ML. To automate memory management, MPL organizes memory into a dynamic tree of heaps called the *heap hierarchy*, which mirrors the spawn tree. Maintenance of the heap hierarchy is illustrated in Figure 2. When a thread is spawned, it receives its own fresh heap in which it allocates all data. When a thread completes, its data is returned to the parent by merging its heap into the parent's.

Disentanglement Definition: With the heap hierarchy, we define disentanglement as the property that threads only "use" data in their root-to-leaf path of heaps. We say that a thread uses some data if the thread is holding a pointer to that data. By maintaining the heap hierarchy, the language runtime automatically ensures that the disentanglement property holds for the generated code [89, 90].

Definition 1 (Disentanglement)**.** A fork-join parallel program is **disentangled** if each thread only holds pointers (in its stack or registers) to data in either its own heap or the heap of an ancestor.

Usefulness of Disentangled Programs: We now compare disentanglement with the classic property of data race freedom (DRF). Previous work leveraged DRF to improve coherence [12, 21, 74, 75]. Here, we show why disentanglement is a more general/useful property for developers.

Disentanglement is more general than data race-freedom because it allows "benign" data races. We describe two examples of disentangled programs that leverage data races.

Our first example is in a prime sieve (described in more detail in §3.3), where multiple threads race to mark numbers as composite; this constitutes a write-write race, but this race is "benign" because all threads are writing the same value.

Next, we consider a parallel breadth-first search of a graph where the search uses inexact criteria (i.e., more than one vertex may meet the search criteria). The threads race to write an acceptable vertex to a shared memory location allocated by the ancestor who initiated the search. It does not matter which thread "wins" the race because they are all writing back values which meet the search criteria.This example highlights that "benign" data races may include write-after-write races with different values.

These examples contain data races, but are nevertheless disentangled. If desired, these algorithms *could* be made entirely data-race-free and deterministic by adding an explicit deduplication step. Instead, disentanglement trades a small amount of non-determinism for fewer allocations and improved performance.

2.2 Cache Coherence

Data Hazards: Disabling coherence exploits the absence or irrelevance of data hazards. Data hazards exist when two hardware threads interact with a shared memory address and at least one thread writes the address. When this occurs, the hardware must order the reads/writes to achieve correctness with respect to the consistency model.[*] Consequently, execution pauses, and overall progress is slowed. Of the three varieties of data hazards (Read-After-Write (RAW), Write-After-Read (WAR), and Write-After-Write (WAW)), WARD reasons about RAW and WAW. Figure 3, Events 1 and 2 give examples of RAW and WAW, respectively. True WAR hazards are prevented by WARDen's reconciliation process (§5.2).

Drawbacks of Cache Coherence: Cache coherence traditionally suffers from inefficiencies. One pitfall is false sharing, which persists as a challenge despite being studied for more than 25 years [87]. Cache coherence protocols also suffer slowdowns due to *true* sharing. Current protocols address all true sharing events equally and reactively. However, some true sharing conflicts, such as benign WAW races found in disentangled programs, do not require fine-grained coherence. In addition, proactively flushing private caches can significantly improve performance (8–28%) [32]. WARDen improves both false sharing and true sharing scenarios to minimize the drawbacks of cache coherence for disentangled programs.

2.3 Cache Coherence for Disciplined Programs

We argue that HLPLs' disciplined memory management enables new advancements to overcome the bottlenecks of cache coherence. Previous works addressed some drawbacks of cache coherence for data-race-free programs [12, 21, 74, 75]. DRF prohibits concurrent accesses to *data* values (i.e., not synchronization primitives) if at least one access is a write. This property is more restrictive than disentanglement because it disallows benign WAW dependencies.

DRF must be enforced on an individual basis for each program. This burden typically falls on the programmer through manual annotation [12, 21, 75]. Requiring the programmer to

[*]We assume Total Store Order (TSO) [63].

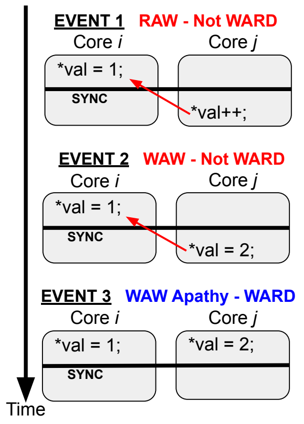


Figure 3. Examples of non-WARD (Events 1, 2) and WARD regions (Event 3). Either value is accepted in Event 3 (§3.2).

carefully annotate the memory behavior of a high-level program defeats the purpose of a memory-managed language.

Disentanglement, on the other hand, is embedded in the language definition. The language runtime ensures that all programs exhibit the property without programmer involvement. By targeting a broader and language-enforced property, WARDen provides a completely transparent improvement for programmers using HLPLs.

3 WARD Property

We now formally define the WARD property and provide examples of how the definition applies to programs, both generally and in a high-level fork-join language.

3.1 WARD Definition

We define the WARD property for a memory location M. M displays the WARD property when two conditions hold for all hardware threads. For any two hardware threads i and j:

1. There exist no execution orders which include RAW dependencies between i and j at M.
2. Any possible WAW dependencies betweeen i and j at M may be resolved in any order.

If these conditions are true for all possible combinations of i and j, then M has the WARD property.

Because the WARD property may exist for a specific set of memory locations and/or for a limited amount of time, we refer to the WARD property in terms of regions. A WARD region r is constrained in memory space and time such that:

$$r = (\{M\}, (t_s, t_e)) \tag{1}$$

During the time interval from t_s to t_e, the set of addresses $\{M\}$ have the WARD property.

3.2 General WARD Example

WARD can be understood by observing the example in Figure 3. Note that the "sync" line in the figure refers to natural synchronization points such as barriers and fork-join points from the fork-join model. We see that each event includes two cores (analogous to the hardware threads in the WARD definition), which both operate on the same variable.

In Event 1, hardware thread i writes to the shared variable *val*. After synchronization, hardware thread j reads and subsequently writes to *val*. When this situation occurs, a RAW dependency exists at *val*. Therefore, the WARD property does not exist by condition 1 of the definition.

In Event 2, i writes to *val*, then j writes to *val*. After synchronization, hardware thread j writes a new value to *val*. When this situation occurs, a WAW dependency exists at *val*. The program requires hardware thread j's final value to persist via the memory fence. The WAW is not apathetic, so the WARD property does not exist by condition 2.

In Event 3, i and j again both write to *val*. We observe that are no RAW dependencies between i and j at *val* because *val* is never read during the event. On the other hand, there is a WAW dependency. However, the program does not provide explicit ordering, so it is safe to resolve the WAW dependency in either order and maintain correctness. Event 3 meets both conditions of the WARD definition, and thus *val* holds the WARD property for the duration of Event 3.

```
// compute all primes less−or−equal to N
// output: array of flags, p is prime if flags[p] is true
def prime_sieve_upto(N):
  bool flags[] = // size N + 1, all initially true
  flags[0] = false
  if N ≥ 1:
    flags[1] = false
  if N ≥ 2:
    bool sqrtflags[] = prime_sieve_upto(⌊√N⌋)
    parallelfor p in range(0, ⌊√N⌋):
      if sqrtflags[p]:
        // p is prime, mark multiples as not prime
        parallelfor m in range(2, ⌊N/p⌋):
          flags[p∗m] = false
  return flags          flags is a WARD region
```

Figure 4. WARD example: prime sieve. Throughout execution, all instances of flags are WARD regions.

3.3 High-Level WARD Example

Figure 4 provides an example of how the WARD property applies to the high-level parallel languages we focus on in this work. The pseudocode is for a prime sieve computation that is a simplified version of the *primes* benchmark we include in our evaluation. It defines a function `prime_sieve_upto` which outputs an array of booleans, `flags`, to mark which integers up to N are prime. For large enough N, the function first computes all primes up to $\lfloor\sqrt{N}\rfloor$ via a recursive call. Then, for each integer p less than $\lfloor\sqrt{N}\rfloor$, if p is prime, it marks every multiple of p as composite. When this process completes, all flags will be correct up to N.

In this example, the flags array is a WARD region. The only races on the flags array are write-write races at indices which are multiples of two or more primes, but the same value (the boolean `false`) is always written at each location. Therefore, the writes may be resolved in any order. Consequently, the flags array satisfies both conditions of the WARD definition.

4 WARD by Construction

We show how WARD regions can be automatically detected in HLPL programs without programmer annotation or compiler analysis. These results are relevant for MPL [2, 61, 90], a compiler for the Parallel ML language.

Conservative Scheduling: For the purposes of connecting logical parallelism (i.e., the fork-join structure) and actual parallelism on the hardware, we assume that the scheduler does not over-parallelize. Specifically, if two instructions occur concurrently on two hardware threads during execution, then the instructions are logically in parallel according to the fork-join structure of the program. All user-level thread schedulers that we know of, including those used in languages and systems such as Cilk, OpenMP, and MPL, follow this assumption. The standard retirement process in an out-of-order superscaler core then assures that effects become visible as the scheduler intends.

4.1 Disentanglement ⇒ WARD

Through the lens of disentanglement and the heap hierarchy (§2.1), we make an immediate observation: at each leaf heap, all data is entirely local to one thread. Therefore, *all leaf heaps are WARD regions in disentangled programs*. This is a critical point because at every moment throughout execution, approximately *half* of all heaps are leaves, even as the hierarchy grows and shrinks due to forks and joins. (Leaf heaps may become internal due a fork, but then later become a leaf again due to a join.)

In this work, we are conservative in that internal heaps *may also* be or contain WARD regions, but we do not leverage them. From disentanglement alone, we cannot ensure that the data at internal heaps is WARD. Disentanglement allows for communication through ancestor (i.e., internal) heaps, which may violate the WARD property.

Despite our conservative assumption, our approach still encompasses many memory accesses, including significant benign WAW races. Allocations occur at leaf heaps, so we can ensure that all newly-allocated data occurs in regions that are WARD in disentangled programs. For programs that utilize considerable immutable data, WARD also covers all memory writes which initialize new immutable objects. Specifically regarding WAWs, significant races can occur from the language runtime interacting with application memory.

4.2 Disentanglement by Construction in MPL

The MPL compiler [2, 61, 90] is a compiler for Parallel ML that extends the Standard ML functional programming language with (nested) fork-join parallelism. MPL directly produces x64 executables (no JIT) linked with a runtime system. To ensure disentanglement by construction, MPL offers a standard library consisting of a number of datatypes, including sequences, sets, dictionaries, etc. The library code is implemented under-the-hood via efficient data structures and algorithms, utilizing in-place updates where crucial for efficiency. This approach allows programmers to write efficient, deterministic, parallel algorithms without needing to reason about either race conditions or disentanglement. The resulting programs are disentangled by construction.

Automatically Exploiting WARD with MPL: In MPL, the runtime system performs task scheduling and automatic memory management. It already exploits disentanglement for improved parallel memory management (especially parallel GC). It does so by explicitly partitioning memory into a dynamic heap hierarchy at runtime.

Again, during execution, all leaf heaps are WARD regions. To take advantage of these WARD regions afforded by disentanglement, we modified two parts of MPL's runtime system: the memory manager and scheduler.

MPL's memory manager implements heaps as linked lists of pages, where allocations are performed within each page via bump-allocation. When a page is exhausted, a fresh page is allocated and the current heap is extended. These pages are always allocated by leaf threads; therefore, whenever a new page is allocated to extend a leaf heap, we mark the page as a WARD region. Pages are later un-marked by the scheduler (which is a standard work-stealing scheduler [11]) at forks, which cause leaf heaps to become internal. At each fork in the program, the scheduler pushes one or more new child tasks onto its work queue and begins working on one of the children with a fresh heap. We modified the scheduler to unmark WARD pages of the current heap before each fork.

4.3 Software Engineering Effort

Our modifications to MPL are invisible to the application programmer. Also, the implementation effort is minor because much of the logic necessary already exists in the memory management of the language run-time. Our implementation in MPL involved adding less than 100 lines of new code.

5 WARDen Cache Coherence Protocol

The WARDen cache coherence protocol augments a standard MESI (Modified, Exclusive, Shared, Invalid) [63] directory-based protocol with a WARD state W. Figure 5 shows the updated directory controller FSA. Cache blocks enter the WARD state when their addresses are contained within WARD regions. When a WARD region is removed, the hardware reconciliation process returns the associated cache blocks to the MESI states. In this paper, we discuss coherence messages as defined by Nagarajan, et al. [63].

5.1 The WARD Coherence State

To begin, the directory tracks all active WARD regions. WARD regions are therefore defined globally. WARD region members transition to the new WARD state when they enter the

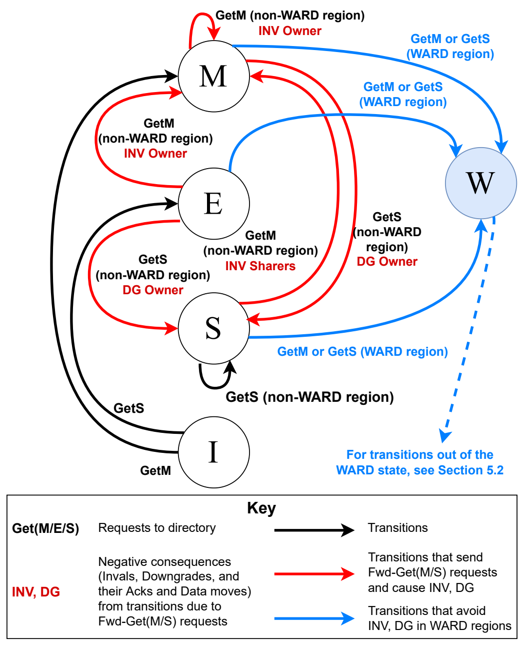

GetM (non-WARD region)
INV Owner
GetM or GetS (WARD region)
GetM or GetS (WARD region)
M
GetM (non-WARD region) INV Owner
W
E
GetM (non-WARD region) INV Sharers
GetS (non-WARD region) DG Owner
GetS (non-WARD region) DG Owner
S
GetM or GetS (WARD region)
GetS (non-WARD region)
GetS
I
GetM
For transitions out of the WARD state, see Section 5.2
Key
Get(M/E/S) Requests to directory
Transitions
INV, DG
Negative consequences (Invals, Downgrades, and their Acks and Data moves) from transitions due to Fwd-Get(M/S) requests
Transitions that send Fwd-Get(M/S) requests and cause INV, DG
Transitions that avoid INV, DG in WARD regions

Figure 5. Simplified WARDen directory FSA. Put(S/E/M/M+data) transitions/transient states not shown. The WARD state (blue) prevents costly downgrades/invalidations (red) during WARD regions.

directory or upon the first sharing event (i.e., a request from a second core). The directory effectively disables coherence for cache blocks in the WARD state. In practice, this means that read and write requests are fulfilled without downgrading or invalidating the same block from other caches.

Shared caches (i.e., caches that are used by multiple cores, such as the LLC) must keep track of the active WARD regions. Using this information, they silently handle any upstream requests from private caches. Similar to the directory, shared caches furnish values in the WARD state without inhibiting the use of other copies. For example, consider a cache hierarchy with private L1 caches and a shared L2. If the L1 cache of core A (L1-A) requests write (or read) access to a cache block currently owned by the L1 of core B (L1-B), the request would flow through L2. If the block is not a WARD region member, L2 would invalidate (or downgrade) the block in L1-B. If the block is in a WARD region, L2 would immediately approve L1-A’s request without bothering L1-B.

When a shared cache receives a GetS (read request) for a WARD block, it returns an exclusive copy to the requestor and hence avoids subsequent upgrade requests even if the same block is concurrently accessed by other cores (e.g., due to false sharing). Thus, WARDen pretends as if the block is private to each core and avoids unnecessary data movement. As a result, private caches can operate as previously defined by the standard cache coherence protocol, and their behavior need not be modified. Leaving private caches unaltered avoids complexity and allows individual cores to operate as quickly as possible. From their perspective, cache blocks in WARD regions are unused by others.

5.2 Reconciliation

When a WARD region is removed, reconciliation merges concurrent updates to cache lines and brings the system to a coherent state. During reconciliation, all WARD cache blocks are placed in the proper MESI state across all cores.

Next, we describe our reconciliation process using three categories: no sharing, false sharing, and true sharing. Note that the directory tracks which cores are sharing each block, and mergers are facilitated by sectored caches (§6).

No Sharing: If only one core is holding a block, it has no sharing. Blocks with no sharing can be instantly converted to the Exclusive state. We are guaranteed program correctness in this case because coherence is fundamentally unnecessary for cache blocks that are not shared.

False Sharing: If a block has multiple sharers that modify distinct sectors, it has false sharing. Blocks with false sharing are merged and set to the Shared state. These blocks were shared by multiple cores, but the individual memory locations within them were not. Therefore, we know that each location was written by at most one hardware thread, and that thread’s local value is the most up-to-date. To reconcile these blocks, we set the global value of each location to the local value from the hardware thread that wrote it. Memory locations that were never written are already consistent.

True Sharing: If a block has multiple sharers that modify the same sector, it has true sharing. True sharing occurs when there is a data hazard between separate cores. We are guaranteed by the WARD property that no RAW hazards will occur, so this form of true sharing is irrelevant. Also, by the WARD property, WAW hazards can be resolved in any order. Thus, we can merge these lines by convenience (e.g., pick the value processed last by the directory).

Note the reconciliation for false and true sharing use the same mechanism. Their distinction is only for understanding.

5.3 Addressing Drawbacks of Cache Coherence

In §2.2, we described inefficiencies in modern coherence protocols. First is false sharing, which we address through the W state. While a cache block maintains state W, loads/stores to the same cache line by other hardware threads are ignored, so false sharing does not lead to coherence traffic.

True sharing presents two improvements. As with false sharing, the W state avoids coherence traffic. Benign WAW hazards are handled by reconciliation, resulting in a one-time cost, which can be overlapped with computation when eviction occurs before the WARD region ends. Additionally, MPL exposes a less obvious software optimization by unmarking pages at forks. Just before forking, a thread writes information required by the new child thread to execute the forked function (e.g., function pointer, input arguments, etc.) into its heap. When the scheduler unmarks these pages, the corresponding cache lines are effectively flushed from the private caches to a shared cache via reconciliation. The hardware-based reconciliation delay is overlapped with the software-based thread creation delay. Then, the newly-created thread immediately accesses these cache lines faster because it is avoiding downgrades to the previous owner's private caches.

All of these optimizations improve application performance by avoiding invalidations (false sharing, unordered WAWs) and downgrades (false sharing, proactive evictions).

6 WARDen System

The WARDen system is our proposed implementation conjoining the WARDen protocol with an HLPL implementation.

6.1 Proposed Hardware Implementation

Implementing the WARDen system requires "Add/Remove Region" instructions, sectored caches, reconciliation logic, and WARD region storage. We combine the WARDen coherence protocol and Parallel ML runtime using two new instructions. The runtime uses these instructions to signal the hardware when a WARD region begins/ends. We expect the addition of two new instructions to have minimal impact.

Sectored caches are necessary for reconciliation to track which memory locations within a cache block are mutated. Sectored caches include additional bits that to track writes at a granularity smaller than the cache block size.

In our proposed implementation, we use byte sectoring to match the smallest granularity in software. This approach adds one bit for every eight data bits. Caches already include substantial metadata including tag bits, coherence state bits, sharer bitmasks in the LLC, and the overhead of SECDED codes for error detection/correction. Using CACTI 7.0 [6], we estimate that byte sectoring on 64-byte cache blocks adds a cache area overhead of 7.9%. We believe our average speedup (1.46x) is far greater than could be gained using the added area less cleverly (e.g., increasing cache size).

Our proposed reconciliation logic is as follows. All WARD cache blocks are flushed (written back as needed and invalidated) from the private caches, and the LLC and directory process requests in the order they arrive. Any sector of a flushed cache block with the write flag set is written back to the shared cache. For the no-sharing and false-sharing cases, no two local copies of a cache block will have the write flag set for the same sector. For the true sharing case, the final value of each sector is taken from whichever cache block is processed last by the LLC; the WARD property guarantees the correctness of this random process.

In practice, we find that WARD regions usually persist long enough to trivialize the reconciliation delay. During evaluation, our prototype reconciled only one block per 50,000 cycles. Also, optimizations outside the scope of this work, such as reconciling blocks in parallel using a multi-banked directory design, could reduce the penalty. For all these reasons,

```
/* Ran on two separate cores (myself and partner) */
while (iterations--) {
  while (buf != partnerID) ;
  buf = myID; }
```

Figure 6. True sharing microbenchmark kernel.

Table 1. Validation of Sniper model (latencies in cycles).

Scenario	Real HW Latency	Simulated Latency
Same core	8.738	11.21
Diff. core, same socket	479.68	286.01
Diff. core, diff. socket	1163.23	1213.59

reconciliation can implemented with reasonable overhead.[†]

Lastly, all directories and shared caches must include storage to track active WARD regions. Regions can be stored with 2 pointers (16 bytes) that indicate their beginning and end. To enable efficient lookups, we model the storage as fully associative caches implemented using CAM-like structures.

To perform a lookup, we use the CAM's per-bit equality comparator to determine the most significant bit that differs from between the region boundary and the address. Then, we check the value of the differing bit. If the address bit is 1, the address is greater than the region boundary. It follows that if the address bit is 0, the address is less than the region boundary. To pass the check, an address must be greater than the lower bound and less than the upper bound.

This logic will require slightly more area than a standard CAM, but it is substantially simpler than the more complicated TCAM since we are not comparing the results across non-paired entries. If an address is somehow found in more than one region, we just mark it as WARD.

Again using CACTI, we estimate supporting 1024 simultaneous regions would require less than 0.05% additional area. This design allows WARD regions to exist for unlimited periods of time, only bounded by the software's directives.

Overall, these results indicate that WARDen is feasible even though we do not claim our implementation is optimal.

6.2 Simulated Prototype

To project the performance improvement and energy savings of the WARDen system, we implement it within the Sniper multicore simulator [16, 17]. We employ the latest version of Sniper [17], which uses an interval simulation technique capable of modeling fine-grained coherence events. We verify that Sniper correctly models the latencies of data movement using a true sharing microbenchmark. We include the kernel of this microbenchmark in Figure 6. The microbenchmark forces a cache block to "ping-pong" between two cores.

Our test system is a two-socket machine with Intel Xeon Gold 6126 processors. Each processor has an L1-L2-L3 (latency in cycles: 6-16-71) cache hierarchy, where L1 and L2 are private and L3 is shared. We configure Sniper identically.

[†]Due to this minimal overhead, our simulator estimates the reconciliation overhead cost by performing a cache flush.

Table 2. Simulated system specifications.

Parameter	Value	Parameter	Value
L1 Size	32 KB	L1/L2 Associativity	8
L2 Size	256 KB	L3 Associativity	20
L3 Size (per core)	2.5 MB	L1/L2/L3 latencies	6-16-71 cycles
Cache Block Size	64 B	Frequency	3.3 GHz
Cores per Socket	12	Intersocket latencies	vary, see text

We evaluate the true sharing microbenchmark in three scenarios, in which the competing hardware threads are on (1) the same core, (2) different cores but the same socket, and (3) different cores and different sockets. We measure the cycles per iteration over 100 million iterations. We run each scenario 10 times and report the measured averages for real and simulated hardware in Table 1. We observe that both the real hardware and simulated latencies align closely. While the simulator latencies are not identical to that of real hardware, they allow for accurate relative comparisons.

7 Evaluation

We compare the performance and energy of standard MPL binaries with the MESI cache coherence protocol versus the WARDen protocol using Sniper. We measure energy consumption using the McPAT [57] power model included with Sniper. Table 2 shows the configuration of the machine and model. We study one and two socket versions and likely future hardware: many-socket and disaggregated systems.

7.1 Evaluation Methodology

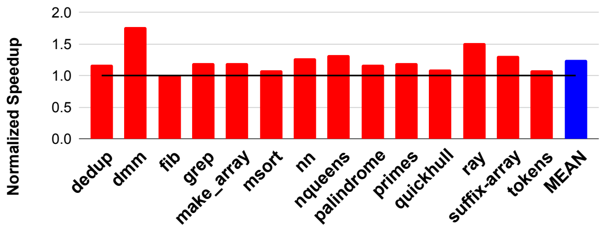
We evaluate the combination of MPL and WARDen using the PBBS benchmark suite [78]. PBBS is a well-cited suite that includes benchmarks from many problem domains, including graph analysis, numerical algorithms, text/image/audio processing, computational geometry, and computer graphics. These benchmarks have been ported to Parallel ML [67] and compiled using MPL, ensuring disentanglement.

We tune benchmark input sizes so each executes without simulation in .1-.5 seconds, resulting in execution times on the simulator between .5-4 hours. We chose these times to feasibly explore multiple configurations.

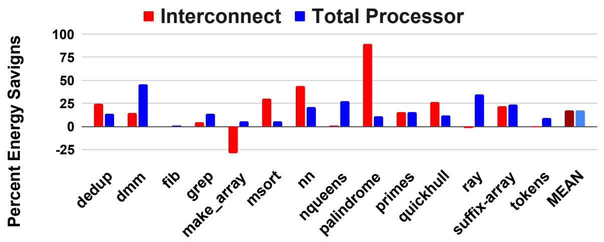
7.2 Modern Hardware

Single socket: We begin with a single socket version of the configuration shown in Table 2. As shown in Figure 7(a), WARDen produces speedups of 1–1.8x, with a mean speedup of 1.24x. Figure 7(b) shows total processor and interconnect energy gains. There is more variation in these, but the averages are 17.4% and 17.3%, respectively. Total processor energy decreases with WARDen in large part because execution time decreases. Network energy decreases due to the smaller number of coherence messages and data transfers. The performance/power overhead of tracking/reconciling WARD regions negatively impacts the results for benchmarks which benefit minimally from WARDen (e.g., make_array).

Dual socket: We now study how WARDen scales to a dual socket system with two of the single-socket processors.

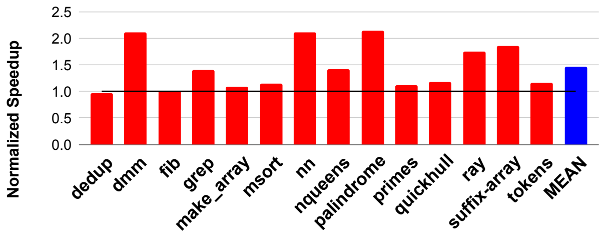


(a) Performance (Speedup)

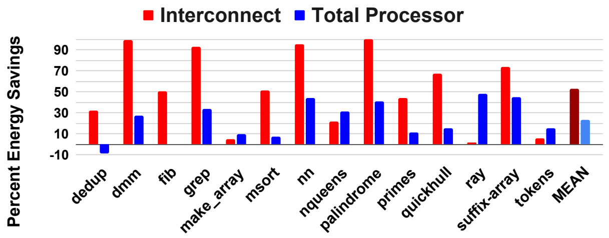


(b) Energy

Figure 7. Performance and energy gains on single socket.



(a) Performance (Speedup)



(b) Energy

Figure 8. Performance and energy gains on dual socket.

Figure 8(a) shows that WARDen produces speedups of 1–2.1x with a mean of 1.46x. These speedups are higher than those from the single socket case, suggesting that the benefits of WARDen scale with machine size. We also see more separation between benchmarks that benefit from WARDen (e.g., palindrome) and those who do not (e.g., dedup).

As shown in Figure 8(b), energy savings increase on the dual socket system compared to the single socket system. For this case, interconnect energy savings, with a mean of 52.9%, outpace the total energy reduction, with a mean of 23.1%, and in some cases, they are the sole driving factor in the cumulative decrease. This result aligns with our expectations because coherence messages are now passed between sockets, therefore consuming far more energy in the network. WARDen is able to eliminate many of these messages. Across the less-accelerated benchmarks, we do not see a negative effect greater than a 5% power increase.

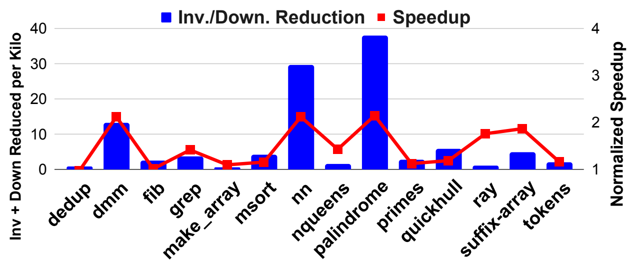


Figure 9. Dual socket speedup with the reduction in invalidations and downgrades.

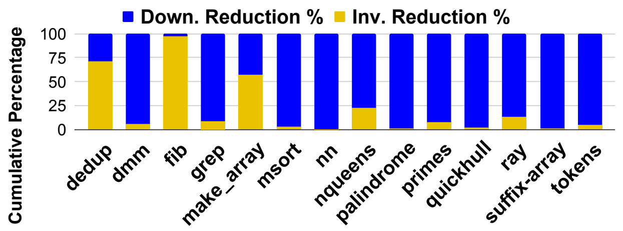


Figure 10. Percent of reduction by invalidations and downgrades.

Analysis: To show that our improvements to cache coherence generate our observed speedups, we study the dual socket results in greater detail. For all the benchmarks except *tokens*, WARDen recognizes the vast majority (90%+) of memory accesses as occurring in a WARD region. There is no correlation with our results because many accesses are to private variables for which there will be no coherence traffic anyway. To understand the effect of WARDen, we focus on the memory accesses that would incur costly downgrades or invalidations with a standard coherence protocol.

In Figure 9, we compare the dual-socket speedups with the reduction in invalidations and downgrades. We count the number of invalidations and downgrades avoided per 1000 instructions executed. Note that invalidations and downgrades are counted per cache, so a single execution may cause many invalidations or downgrades throughout the cache hierarchy. Figure 9 shows a positive correlation between reducing costly memory events and speedup. For many benchmarks, WARDen avoids invalidations and downgrades, which in turn accelerates performance. Conversely, benchmarks with small reductions of these events show little speedup.

A few measurements in Figure 9 appear anomalous. Three benchmarks (*nqueens*, *ray*, and *suffix_array*) show significant speedups, yet have relatively small reductions in coherence events. Meanwhile, *fib*, *msort*, *primes*, and *quickhull* underperform given what might be expected given their relatively large reductions in coherence events.

To dive deeper, we show the breakdown of invalidations and downgrades by percentage in Figure 10. Downgrades are generally more important than invalidations for application performance because they affect load operations. Loads are *blocking* operations that pause dependent computation. In contrast, invalidations are caused by store operations,

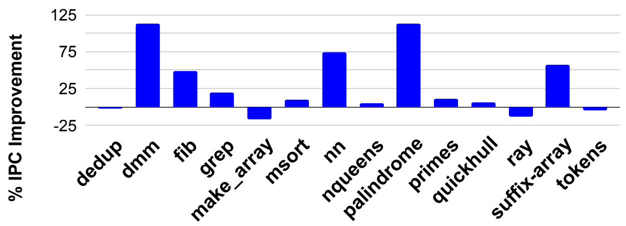


Figure 11. Percentage IPC improvement.

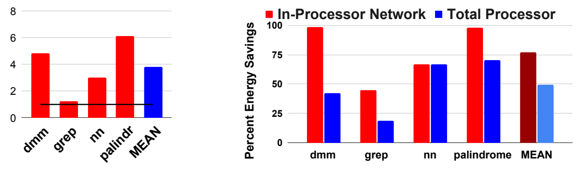
which are injected into the processor's store buffer and commit without waiting for their completion in the memory hierarchy. Unless the store buffer is full, which is relatively rare, store latency (and hence invalidation latency) does not impact execution. Figure 10 shows that *Nqueens*, *ray*, and *suffix_ray* mostly avoid downgrades (77.7%, 86.4%, and 98.3%, respectively). Contrast these results with a more subtle outlier, *fib*. *Fib* experiences a significant reduction in negative coherence events but does not see any appreciable speedup. This likely occurs because *fib* has the lowest percentage of downgrades out of all benchmarks (2.65%).

Now, we explain the results for *msort*, *primes*, and *quickhull*. All three mostly avoid downgrades, so we would expect more substantial speedups. However, the performance of these benchmarks is not bound by coherence events. To understand these benchmarks, we plot the percent IPC improvement generated by WARDen in Figure 11. IPC helps us to understand the application's ability to continue executing instructions despite coherence delays. Benchmarks with low IPC improvements do not take advantage of the faster memory accesses provided by WARDen, indicating their speedup should be lower. Surely enough, all of *msort*, *primes*, and *quickhull* show minimal IPC improvement from avoiding downgrades and invalidations.

Perhaps the most interesting measurement in Figure 11 is *ray*, which shows an IPC reduction despite its large speedup. We believe this IPC result indicates an improvement to synchronization delays. The PBBS benchmark suite uses busy-waiting synchronization primitives implemented using *compare_and_swap* atomics. Busy waiting involves executing many cheap read/write instructions. We theorize that WARDen's improvements help individual threads reach synchronization points more quickly and evenly, eliminating fast waiting instructions, and thus lowering the IPC despite improving application performance. We observe a 49.5% reduction in load instructions executed by *ray* that justifies this claim. This reduction of instructions executed also further supports the reported speedup. We continue to study these benchmarks in detail to completely understand their complex interactions with WARDen.

7.3 Future Machines

Many Sockets: In the future, we expect to see systems with many more sockets become more commonplace. Programs



(a) Speedup **(b)** Energy

Figure 12. Performance and energy gains on disaggregated.

written in HLPLs are a great candidate to run on such machines because they allow expression of algorithmic parallelism, making readily available the higher levels of parallelism such machines require. In this case, interconnect latencies (like those in Table 1) will continue to rise. We expect the advantages of WARDen to become even more prevalent in such a situation. To better understand why, we consider another likely future hardware scheme: disaggregation.

Disaggregated: To model a disaggregated system, we emulate a 2 node machine with a remote access time of 1 μs. This time is conservative, outstripping the performance of state of the art systems [15, 36]. We include the most promising benchmarks from our study of modern hardware.

Figure 12 shows that benchmark speedup improves dramatically, to about a mean of 3.8×, when we disaggregate the two processors from their shared memory hierarchy. As seen in Figure 12, network energy savings improves to a mean of ~77.1% and processor energy savings improves to a mean of ~49.5%. This result makes sense because our disaggregated system has a L3 miss penalty more than 3× greater than our standard dual socket system. Coherence downgrades and flushes are therefore more costly, which in turn makes WARDen more valuable.

A subtle result of this experiment is that disaggregation appears to widen the performance-improvement gap between benchmarks that drastically benefit from WARDen and those with more middling results. *Grep* was the weakest performer of the selected benchmarks in the experiments simulating standard hardware, and its speedup did not grow nearly as much as the others in the disaggregated case.

8 Related Work

Removing/deactivating coherence has been proposed before in hardware-supported, compiler-directed (HSCD) cache coherence [23], OS-driven coherence deactivation [24], and software cache coherence [5, 20, 22, 25, 51, 64, 66, 85, 86]. Our work is distinct because we drive coherence deactivation from the properties of HLPLs, which can provide the information to control coherence by construction. In contrast, these prior works give this task to the programmer (pragmas) or recover this information through run-time inspector-executor methods and compiler analyses. These analyses treat entire arrays as single variables and fail to detect false sharing [20, 22], or limit array subscripts to loop iterators [25]. Others rely on software for triggering coherence actions and hardware for selective self-invalidations but incur high overhead in lock-intensive programs [5] or are restricted to unity loop iterators in affine loops without conditionals [64].

Other works improve cache coherence for discplined memory programs. DeNovo [21, 84] simplifies hardware cache coherence by banning "wild shared-memory behaviors", but it uses a restrictive programming model that requires user-annotated code. VIPS-M [75] avoids directory accesses and invalidations given data-race-free guarantees from software. This approach is limited compared to WARDen because it only supports DRF programs. In addition, it does not support legacy applications, meaning all programs must enforce DRF. SPEL expands VIPS by implementing a dual cache coherence protocol, allowing for legacy, non-DRF applications [74]. SPEL relies on static compiler analysis to identify DRF code regions, which limits its scope compared to WARDen. The static analysis will fail on any disentangled, non-DRF regions. Also, WARDen avoids any compile-time overhead/analysis. Jimborean et al [47] recognize DRF regions in programs manually parallelized with *pthreads* using static compiler analysis and target the SPEL dual cache coherence protocol. The performance of this approach is limited by the conservativeness of modern alias/pointer-analysis; it is unable to detect up to 50% of potential extended DRF regions. In contrast, WARDen avoids the limitations of compile-time analysis by targeting the disentanglement property of HLPLs. For all these DRF-based works, note that disentanglement encompasses DRF. Therefore, these works could be tweaked to target WARDen, allowing WARDen to support some non-HLPL programs.

Alternative cache designs and protocols [18, 28, 43, 50, 51, 73] and transactional memory [40, 41, 77] also relax hardware coherence according to higher-level directives.

9 Conclusion

Lower-level parallel languages have been implicitly and explicitly co-designed with hardware to varying degrees for decades. With HLPLs on the rise, it is time to explore new co-design opportunities they create. We have shown that it is uniquely possible to *automatically* reign in the cost of cache coherence for HLPLs without burdening the programmer. The artifact for this work is available online [91].

Acknowledgments

This research was partially supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357, and by the U.S. National Science Foundation via awards CNS-1763743, CCF-2028851, CCF-2119069, CCF-2115104, CCF-2119352, CCF-1901381, CCF-2107241, CCF-2107042, CCF-1908488, and CCF-2118708.

A Artifact Appendix

A.1 Abstract

This artifact is a virtual machine of Red Hat Enterprise Linux (RHEL) containing the WARDen prototype and its dependencies. The artifact is pre-installed in the "cgo_artifact" account. The password is the same as the username: cgo_artifact. All the PBBS benchmarks used in the paper are also included. This artifact requires VMware Workstation 17 player to load and run the VM, which can be freely downloaded online.

A.2 Artifact Check-List (Meta-Information)

- **Algorithm: No**
- **Program: PBBS, Included**
- **Compilation: MPL, Included**
- **Transformations: No**
- **Binary: No**
- **Data set: Included with the benchmark suite**
- **Run-time environment: No**
- **Hardware: None (results are simulated using Sniper 7.3)**
- **Software: MLton (release 20210117), and Sniper 7.3 and its dependencies (pin 3.13 & mcpat 1.0), as well as versions of any software used by MPL (gcc, ld, etc.). These dependencies are included in the virtual machine.**
- **Run-time environment: N/A**
- **Execution: Sole user, approximately 4 days to run**
- **Metrics: Simulated execution time, other architectural statistics**
- **Output: Command line text output for each benchmark**
- **Experiments: The experiments can be run using included scripts. Detailed instructions are included in the following sections.**
- **How much disk space required (approximately)?: 40GB**
- **How much time is needed to prepare workflow (approximately)?: A few hours**
- **How much time is needed to complete experiments (approximately)?: 4 days**
- **Publicly available?: Yes**
- **Code licenses (if publicly available)?: HPND (MPL), MIT + Interval Academic License (Sniper)**
- **Data licenses (if publicly available)?: No**
- **Workflow framework used?: Complilation and benchmarking is automated using scripts and *make*.**
- **Archived (provide DOI)?: 10.5281/zenodo.7374334**

A.3 Description

A.3.1 How Delivered

A compressed folder containing the VM can be downloaded from the Zenodo at: https://doi.org/10.5281/zenodo.7374334 .

A.3.2 Hardware Dependencies

The artifact must be run on some hardware capable of running the VM. The amount of disk space required by the VM is approximately 40 GB.

A.3.3 Software Dependencies

To run the artifact, VMware Workstation 17 Player is required. All software related to the paper comes pre-installed in the provided VM. We specifically tested with version 17.0.0 build-20800274 running on Ubuntu 20.04.

A.3.4 Data Sets

The input data sets for the benchmarks are already included in the VM.

A.4 Installation

Download the VM compressed folder and VMware Workstation 17 Player. Extract the compressed folder at the top level of VMware's directory. Open VMware Player and select "Open a Virtual Machine". Select the newly-extracted folder. This will add a RHEL VM to the left side. Power on the VM, and choose "I copied it" when prompted. The VM should now boot to the login screen. Login to the "cgo_artifact" account. The password is the same as the username: cgo_artifact. Open a terminal using the "Activities" tab in the top left corner.

A.5 Experiment Workflow

After loading the VM, enter the *sniper_coherence/test* directory to perform experiments. In this directory, there is a file named *ENV* that sets important environment variables. At the beginning of each new session, run *source ENV* to establish the environment. Then, using simple *make* commands, evaluators can reproduce the experiments from our paper. The following workflow is automatically executed when invoking *make all_pbbs* in the VM.

1. Each benchmark is run using the architectural simulator twice: once with the standard coherence protocol, and once with the WARDen protocol.
2. Simulated execution time/power data (Figures 7/8) and statistics regarding the number of downgrades/invalidations and IPC (Figures 9/10/11) are all printed to the command line.

A.6 Evaluation and Expected Result

To reproduce the experiments, there are four relevant commands:

- *make single_pbbs*: This command runs a single benchmark. To specify the benchmark, set the *BENCH* value to the name of the benchmark. Example usage: *make single_pbbs BENCH=fib* runs the fib benchmark.
- *make all_pbbs*: This command runs all the PBBS benchmarks used in this paper. This command has no arguments.
- *make activate_one_socket*: This command switches the simulator to the one-socket configuration, as used in Figure 7.
- *make activate_two_socket*: This command switches the simulator to the two socket configuration, as used in Figure 8 and following analysis.

To reproduce all the single-socket and two-socket experiments from the paper, use the following order:

1. *cd sniper_coherence/test*
2. *source ENV*
3. *make activate_one_socket*
4. *make all_pbbs*
5. *make activate_two_socket*
6. *make all_pbbs*

We expect the speedups and power results to align very closely to Figures 7 and 8.

A.7 Experiment Customization

The primary way to customize our experiments is by changing the input values for the benchmarks. Each benchmark has its own directory under *sniper_coherence/test/pbbs* with individual READMEs that include suggested input sizes. To change the input, edit the benchmark's Makefile.

Our workflow can be customized to evaluate different applications, assuming they are compatible with our software framework (e.g., written in Parallel ML). To add a new benchmark, simply create a new folder under *pbbs* with a Makefile matching the form of the others, and all the existing scripts will work.

References

[1] Sarita V Adve and Mark D Hill. 1990. Weak ordering—a new definition. *ACM SIGARCH Computer Architecture News* 18, 2SI (1990), 2–14.

[2] Jatin Arora, Sam Westrick, and Umut A. Acar. 2021. Provably Space Efficient Parallel Functional Programming. In *Proceedings of the 48th Annual ACM Symposium on Principles of Programming Languages (POPL)"*.

[3] Akhil Arunkumar, Evgeny Bolotin, Benjamin Cho, Ugljesa Milic, Eiman Ebrahimi, Oreste Villa, Aamer Jaleel, Carole-Jean Wu, and David Nellans. 2017. MCM-GPU: Multi-chip-module GPUs for continued performance scalability. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 320–332. https://doi.org/10.1145/3079856.3080231

[4] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. 1989. I-structures: Data Structures for Parallel Computing. *ACM Trans. Program. Lang. Syst.* 11, 4 (Oct. 1989), 598–632.

[5] Thomas J. Ashby, Pedro Díaz, and Marcelo Cintra. 2011. Software-Based Cache Coherence with Hardware-Assisted Selective Self-Invalidations Using Bloom Filters. *IEEE Trans. Comput.* 60, 4 (2011), 472–483. https://doi.org/10.1109/TC.2010.155

[6] Rajeev Balasubramonian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 2 (2017), 1–25.

[7] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Charles E. Leiserson. 2004. On-the-Fly Maintenance of Series-Parallel Relationships in Fork-Join Multithreaded Programs. In *16th Annual ACM Symposium on Parallel Algorithms and Architectures*. 133–144.

[8] Guy E. Blelloch. 1996. Programming Parallel Algorithms. *Commun. ACM* 39, 3 (1996), 85–97.

[9] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. 2012. Internally deterministic parallel algorithms can be fast *(PPoPP '12)*. 181–192.

[10] Guy E. Blelloch, Jonathan C. Hardwick, Jay Sipelstein, Marco Zagha, and Siddhartha Chatterjee. 1994. Implementation of a Portable Nested Data-Parallel Language. *J. Parallel Distrib. Comput.* 21, 1 (1994), 4–14.

[11] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *J. ACM* 46, 5 (Sept. 1999), 720–748.

[12] Robert L Bocchino Jr, Vikram S Adve, Danny Dig, Sarita V Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009. A type and effect system for deterministic parallel Java. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. 97–116.

[13] Bull. 2021. Bull Bullion S16 Server. http://www.scaleupservers.com/Bullion-S16-Server.asp.

[14] Paul Caheny, Lluc Alvarez, Said Derradji, Mateo Valero, Miquel Moretó, and Marc Casas. 2018. Reducing Cache Coherence Traffic with a NUMA-Aware Runtime Approach. *IEEE Transactions on Parallel and Distributed Systems* 29, 5 (2018), 1174–1187. https://doi.org/10.1109/TPDS.2017.2787123

[15] 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 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 79–92. https://doi.org/10.1145/3445814.3446713

[16] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. 2011. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulations. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[17] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. 2014. An Evaluation of High-Level Mechanistic Core Models. *ACM Transactions on Architecture and Code Optimization (TACO)*, Article 5 (2014), 23 pages. https://doi.org/10.1145/2629677

[18] Nicholas P. Carter, Aditya Agrawal, Shekhar Borkar, Romain Cledat, Howard David, Dave Dunning, Joshua Fryman, Ivan Ganev, Roger A. Golliver, Rob Knauerhase, Richard Lethin, Benoit Meister, Asit K. Mishra, Wilfred R. Pinfold, Justin Teller, Josep Torrellas, Nicolas Vasilache, Ganesh Venkatesh, and Jianping Xu. 2013. Runnemede: An architecture for Ubiquitous High-Performance Computing. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. 198–209. https://doi.org/10.1109/HPCA.2013.6522319

[19] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. 1998. Detecting data races in Cilk programs that use locks. In *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures (SPAA '98)*.

[20] H. Cheong and A.V. Veidenbaum. 1988. A cache coherence scheme with fast selective invalidation. In *[1988] The 15th Annual International Symposium on Computer Architecture. Conference Proceedings*. 299–307. https://doi.org/10.1109/ISCA.1988.5240

[21] Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V Adve, Vikram S Adve, Nicholas P Carter, and Ching-Tsun Chou. 2011. DeNovo: Rethinking the memory hierarchy for disciplined parallelism. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 155–166.

[22] L. Choi and Pen-Chung Yew. 1994. A compiler-directed cache coherence scheme with improved intertask locality. In *Supercomputing '94:Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*. 773–782. https://doi.org/10.1109/SUPERC.1994.344343

[23] Lynn Choi and Pen-Chung Yew. 1996. Compiler and Hardware Support for Cache Coherence in Large-Scale Multiprocessors: Design Considerations and Performance Study. In *Proceedings of the 23rd annual international symposium on Computer architecture*.

[24] Blas Cuesta, Alberto Ros, María E Gómez, Antonio Robles, and José Duato. 2011. Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 93–103.

[25] Ervan Darnell, John M. Mellor-Crummey, and Ken Kennedy. 1992. Automatic Software Cache Coherence through Vectorization. In *Proceedings of the 6th International Conference on Supercomputing* (Washington, D. C., USA) *(ICS '92)*. 129–138. https://doi.org/10.1145/143369.143398

[26] Abhishek Das, Matt Schuchhardt, Nikos Hardavellas, Gokhan Memik, and Alok Choudhary. 2012. Dynamic Directories: A mechanism for reducing on-chip interconnect power in multicores. In *Proceedings of the Conference on Design, Automation Test in Europe* (Dresden, Germany). 479–484.

[27] Yigit Demir, Yan Pan, Seukwoo Song, Nikos Hardavellas, John Kim, and Gokhan Memik. 2014. Galaxy: A High-Performance Energy-Efficient Multi-Chip Architecture Using Photonic Interconnects. In *Proceedings*

of the 28th ACM International Conference on Supercomputing (Munich, Germany) *(ICS'14)*.

[28] Marco Elver and Vijay Nagarajan. 2014. TSO-CC: Consistency directed cache coherence for TSO. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 165–176.

[29] Natalie Enright Jerger, Li-Shiuan Peh, and Mikko Lipasti. 2008. Virtual Tree Coherence: Leveraging Regions and In-Network Multicast Trees for Scalable Cache Coherence. In *International Symposium on Microarchitecture*. 35–46.

[30] Ericsson. 2017. Time for memory disaggregation? https://www.ericsson.com/en/blog/2017/5/time-for-memory-disaggregation.

[31] Mingdong Feng and Charles E. Leiserson. 1999. Efficient Detection of Determinacy Races in Cilk Programs. *Theory of Computing Systems* 32, 3 (1999), 301–326.

[32] Sevin Fide and Stephen Jenks. 2008. Proactive use of shared L3 caches to enhance cache communications in multi-core processors. *IEEE Computer Architecture Letters* 7, 2 (2008), 57–60.

[33] Jeremy T. Fineman. 2005. *Provably Good Race Detection That Runs in Parallel*. Master's thesis. Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge, MA.

[34] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. 2011. Implicitly threaded parallelism in Manticore. *Journal of Functional Programming* 20, 5-6 (2011), 1–40.

[35] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. 2009. Reducers and Other Cilk++ Hyperobjects. In *21st Annual ACM Symposium on Parallelism in Algorithms and Architectures*. 79–90.

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

[37] Adrien Guatto, Sam Westrick, Ram Raghunathan, Umut A. Acar, and Matthew Fluet. 2018. Hierarchical memory management for mutable state. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018*. 81–93.

[38] Robert H. Halstead, Jr. 1984. Implementation of Multilisp: Lisp on a Multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming* (Austin, Texas, United States) *(LFP '84)*. ACM, 9–17.

[39] Kevin Hammond. 2011. Why Parallel Functional Programming Matters: Panel Statement. In *Reliable Software Technologies - Ada-Europe 2011 - 16th Ada-Europe International Conference on Reliable Software Technologies, Edinburgh, UK, June 20-24, 2011. Proceedings*. 201–205.

[40] Tim Harris, James Larus, and Ravi Rajwar. 2010. Transactional memory. *Synthesis Lectures on Computer Architecture* 5, 1 (2010), 1–263.

[41] Maurice Herlihy and J Eliot B Moss. 1993. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on Computer architecture*. 289–300.

[42] Hewlett Packard Enterprise. 2021. HPE Integrity MC990 X Server. https://www.hpe.com/psnow/doc/PSN1008798952USEN.pdf.

[43] Derek R. Hower. 2012. *Acoherent Shared Memory*. Ph. D. Dissertation. USA. Advisor(s) Hill, Mark D. AAI3522117.

[44] C.C. Hu, M.F. Chen, W.C. Chiou, and Doug C.H. Yu. 2019. 3D Multi-chip Integration with System on Integrated Chips (SoIC™). In *2019 Symposium on VLSI Technology*. T20–T21. https://doi.org/10.23919/VLSIT.2019.8776486

[45] IBM. 2018. Advancing cloud with memory disaggregation. https://www.ibm.com/blogs/research/2018/01/advancing-cloud-memory-disaggregation/.

[46] Subramanian S. Iyer. 2016. Heterogeneous Integration for Performance and Scaling. *IEEE Transactions on Components, Packaging and Manufacturing Technology* 6, 7 (2016), 973–982. https://doi.org/10.1109/TCPMT.2015.2511626

[47] Alexandra Jimborean, Jonatan Waern, Per Ekemark, Stefanos Kaxiras, and Alberto Ros. 2017. Automatic detection of extended data-race-free regions. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 14–26.

[48] Robert L. Bocchino Jr., Stephen Heumann, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Adam Welc, and Tatiana Shpeisman. 2011. Safe nondeterminism in a deterministic-by-default parallel language. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 535–548.

[49] Ajaykumar Kannan, Natalie Enright Jerger, and Gabriel Loh. 2015. Enabling Interposer-based Disintegration of Multi-core Processors. In *Proceedings of the International Symposium on Microarchitecture*.

[50] Pete Keleher, Alan L Cox, and Willy Zwaenepoel. 1992. Lazy release consistency for software distributed shared memory. *ACM SIGARCH Computer Architecture News* 20, 2 (1992), 13–21.

[51] Wooil Kim, Sanket Tavarageri, P. Sadayappan, and Josep Torrellas. 2016. Architecting and Programming a Hardware-Incoherent Multiprocessor Cache Hierarchy. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 555–565. https://doi.org/10.1109/IPDPS.2016.76

[52] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Konstantin Taranov, Dejan S. Milojicic, and Gustavo Alonso. 2021. Farview: Disaggregated Memory with Operator Off-loading for Database Engines. *CoRR* abs/2106.07102 (2021). arXiv:2106.07102 https://arxiv.org/abs/2106.07102

[53] Lindsey Kuper and Ryan R Newton. 2013. LVars: lattice-based data structures for deterministic parallelism. In *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing*. ACM, 71–84.

[54] Lindsey Kuper, Aaron Todd, Sam Tobin-Hochstadt, and Ryan R. Newton. 2014. Taming the Parallel Effect Zoo: Extensible Deterministic Parallelism with LVish. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. 2–14.

[55] Youngeun Kwon and Minsoo Rhu. 2019. A Disaggregated Memory System for Deep Learning. *IEEE Micro* 39, 5 (2019), 82–90. https://doi.org/10.1109/MM.2019.2929165

[56] Peng Li, Simon Marlow, Simon L. Peyton Jones, and Andrew P. Tolmach. 2007. Lightweight concurrency primitives for GHC. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2007, Freiburg, Germany, September 30, 2007*. 107–118.

[57] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *MICRO*.

[58] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. 2009. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture* (Austin, TX, USA) *(ISCA '09)*. 267–278. https://doi.org/10.1145/1555754.1555789

[59] John Mellor-Crummey. 1991. On-the-fly Detection of Data Races for Programs with Nested Fork-Join Parallelism. In *Proceedings of Supercomputing'91*. 24–33.

[60] Moor Insights and Strategy. 2013. Intel's Disaggregated Server Rack. https://moorinsightsstrategy.com/wp-content/uploads/2013/08/Intels-Disagggregated-Server-Rack-by-Moor-Insights-Strategy.pdf.

[61] MPL [n. d.]. MPL compiler. https://github.com/mpllang/mpl.

[62] Samuel Naffziger, Noah Beck, Thomas Burd, Kevin Lepak, Gabriel H. Loh, Mahesh Subramony, and Sean White. 2021. Pioneering Chiplet

Technology and Design for the AMD EPYC™ and Ryzen™ Processor Families : Industrial Product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 57–70. https://doi.org/10.1109/ISCA52012.2021.00014

[63] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2020. *A Primer on Memory Consistency and Cache Coherence: Second Edition.*

[64] M. F. P. O'Boyle, R. W. Ford, and E. A. Stohr. 2003. Towards General and Exact Distributed Invalidation. *J. Parallel Distrib. Comput.* 63, 11 (Nov. 2003), 1123–1137. https://doi.org/10.1016/j.jpdc.2003.07.007

[65] OpenMP 5.0 2018. *OpenMP Application Programming Interface, Version 5.0.* Accessed in July 2018.

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

[67] Parallel ML Benchmarks [n. d.]. https://github.com/mpllang/parallel-ml-bench.

[68] Ivy Peng, Roger Pearce, and Maya Gokhale. 2020. On the Memory Underutilization: Exploring Disaggregated Memory on HPC Systems. In *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 183–190. https://doi.org/10.1109/SBAC-PAD49847.2020.00034

[69] Simon L. Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. 2008. Harnessing the Multicores: Nested Data Parallelism in Haskell. In *FSTTCS*. 383–414.

[70] Ram Raghunathan, Stefan K. Muller, Umut A. Acar, and Guy Blelloch. 2016. Hierarchical Memory Management for Parallel Programs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP 2016)*. ACM, New York, NY, USA, 392–406.

[71] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. 2010. Efficient Data Race Detection for Async-Finish Parallelism. In *Runtime Verification*, Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann (Eds.). Lecture Notes in Computer Science, Vol. 6418. Springer Berlin / Heidelberg, 368–383.

[72] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. 2012. Scalable and Precise Dynamic Datarace Detection for Structured Parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. 531–542.

[73] Yuxin Ren, Gabriel Parmer, and Dejan Milojicic. 2020. Ch'i: Scaling Microkernel Capabilities in Cache-Incoherent Systems. In *2020 IEEE/ACM International Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*. 12–21. https://doi.org/10.1109/ROSS51935.2020.00007

[74] Alberto Ros and Alexandra Jimborean. 2015. A dual-consistency cache coherence protocol. In *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 1119–1128.

[75] Alberto Ros and Stefanos Kaxiras. 2012. Complexity-effective multicore coherence. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 241–251.

[76] Matthew Schuchhardt, Abhishek Das, Nikos Hardavellas, Gokhan Memik, and Alok Choudhary. 2013. The Impact of Dynamic Directories on Multicore Interconnects. *IEEE Computer* 46, 10 (October 2013), 32–39.

[77] Nir Shavit and Dan Touitou. 1997. Software transactional memory. *Distributed Computing* 10, 2 (1997), 99–116.

[78] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. 2012. Brief Announcement: The Problem Based Benchmark Suite *(SPAA '12)*. 68–70. https://doi.org/10.1145/2312005.2312018

[79] KC Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. 2020. Retrofitting parallelism onto ocaml. *arXiv preprint arXiv:2004.11663* (2020).

[80] KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting effect handlers onto OCaml. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 206–221.

[81] K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. 2020. Retrofitting parallelism onto OCaml. *Proc. ACM Program. Lang.* 4, ICFP (2020), 113:1–113:30.

[82] Daniel Spoonhower. 2009. *Scheduling Deterministic Parallel Programs.* Ph. D. Dissertation. Carnegie Mellon University. https://www.cs.cmu.edu/~rwh/theses/spoonhower.pdf

[83] Rabin Sugumar, Mehul Shah, and Ricardo Ramirez. 2021. Marvell ThunderX3: Next-Generation Arm-Based Server Processor. *IEEE Micro* 41, 2 (2021), 15–21. https://doi.org/10.1109/MM.2021.3055451

[84] Hyojin Sung, Rakesh Komuravelli, and Sarita V. Adve. 2013. DeNovoND: Efficient Hardware Support for Disciplined Non-Determinism. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. 13–26. https://doi.org/10.1145/2451116.2451119

[85] Igor Tartalja and Veljko Milutinovic. 1996. The Cache Coherence Problem in Shared Memory Multiprocessors: Software Solutions. In *XVI International Symposium on Nuclear Electronics and VI International School on Automation and Computing in Nuclear Physics and Astrophysics*. 131.

[86] Sanket Tavarageri, Wooil Kim, Josep Torrellas, and P Sadayappan. 2016. Compiler support for software cache coherence. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*. IEEE, 341–350.

[87] Josep Torrellas, HS Lam, and John L. Hennessy. 1994. False sharing and spatial locality in multiprocessor caches. *IEEE Trans. Comput.* 43, 6 (1994), 651–663.

[88] Robert Utterback, Kunal Agrawal, Jeremy T. Fineman, and I-Ting Angelina Lee. 2016. Provably Good and Practically Efficient Parallel Race Detection for Fork-Join Programs. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*. 83–94.

[89] Sam Westrick, Jatin Arora, and Umut A. Acar. 2022. Entanglement detection with near-zero cost. *Proc. ACM Program. Lang.* 6, ICFP (2022), 679–710. https://doi.org/10.1145/3547646

[90] Sam Westrick, Rohan Yadav, Matthew Fluet, and Umut A. Acar. 2020. Disentanglement in Nested-Parallel Programs. In *Proceedings of the 47th Annual ACM Symposium on Principles of Programming Languages (POPL)"*.

[91] Michael Wilkins, Sam Westrick, Vijay Kandiah, Alex Bernat, Brian Suchy, Enrico Armenio Deiana, Simone Campanoni, Umut Acar, Peter Dinda, and Nikos Hardavellas. 2022. Artifact for "WARDen: Specializing Cache Coherence for High-Level Parallel Languages". https://doi.org/10.5281/zenodo.7374334

Received 2022-09-02; accepted 2022-11-07