2018 IEEE International Parallel and Distributed Processing Symposium

Architectural support for unlimited memory versioning and renaming

Eran Gilad*, Tehila Mayzels*, Elazar Raab*, Mark Oskin† and Yoav Etsion*
*Technion — Israel Institute of Technology
erangi@cs.technion.ac.il
yetsion@tce.technion.ac.il
†University of Washington
oskin@cs.washington.edu

***Abstract*—Data versioning and renaming is a technique to enforce true dependencies and eliminate false dependencies in concurrent out-of-order execution. By extending the addressing to memory to support both a location and a version number, the memory system can match loads with the appropriate stores. With multiple versions of data for a single memory location, Write-after-Read and Write-after-Write dependencies are avoided.**

In this paper, we present architectural support for O-structures, which provide memory versioning and renaming. We describe a microarchitectural implementation of an O-structure in the cache hierarchy of a multicore processor and demonstrate the need of each feature provided by O-structures. Our evaluation shows that O-structures can be effective in supporting a range of parallel workloads, including irregular, pointer-heavy code.

***Keywords*-Memory Systems; Hardware/Software interface; Multicore/Parallel Architectures; Parallel programming; Computer Architecture**

I. Introduction

Data versioning and renaming has proven to be extremely useful in multiple scenarios. At the micro-architectural level, registers are renamed to eliminate false dependencies, and mechanisms such as ARB [1] and Task Superscalar [2] have achieved the same goal with memory addresses. Speculative execution mechanisms such as TLS and transactional memory use versioning and renaming to order concurrent operations and provide snapshot isolation. Similar needs have been identified and handled in software, whether frameworks (e.g., STM [3]) or data structures (e.g., MVBT [4]).

While versioning and renaming is a common tool, most uses involve a tailored solution, suiting their specific needs. The efficiency of those solution usually comes from limiting the use of resources, restricting the number of either the versions per address, the addresses renamed, or both. Such limitations prohibit the reuse of already proposed mechanisms. Moreover, hardware mechanisms have not made versioning and renaming architecturally visible, excluding their use by software.

This paper describes architectural support for memory versioning and renaming using O-structures [5] and proposes a micro-architectural implementation. Our implementation transparently manages the physical storage of both data and metadata (versions), and provide various access semantics, including versioned stores, loads and locks. Further, the number of memory locations that can be renamed and versions per location is unbound, limited only by the amount of RAM dedicated by the system at run time. Our O-structures implementation provides an efficient and fast solution to software requiring memory versioning and renaming.

The proposed implementation is integrated into the caching subsystem, without modifying the main memory. Adding versioning support to the caches allow recently used versions to be directly accessed. In case of a miss, internal version linking is used to perform a full lookup. While versioning adds overhead, our hardware implementation minimizes caching footprint by compressing metadata and selectively caching versions accessed during full lookups. Internal versions linking is isolated from user code, keeping the security guarantees provided by virtual memory.

To demonstrate the full range of capabilities, we present 3 different use cases. We start by showing how versioning can order execution of regular operations such as matrix multiplication, where data must not be consumed before it is produced, yet only a single version of each datum is required. Then, we show how renaming eliminates false dependencies and provides snapshot isolation, yielding higher parallelism and throughput than a conventional read-write lock. Finally, we show the full extent of O-structure features by parallelizing irregular operations, which leverages fine-grained locking to maintain ordering even when dependencies are not known.

This paper makes the following contributions:

- Define a memory interface that exposes versioning, renaming and fine-grained locking
- Propose a micro-architectural implementation that does not bound the number of versioned addresses or renames, yet does not affect conventional memory use.
- Demonstrate the use of the extended memory system in widespread codes.

II. O-structures

A. Semantics and instruction set

An O-structure is a memory element that maintains multiple versions of the datum. These versions are ordered

Prog. model	Versioning	Purpose	Exec. model	Versioning used by
Functional	Full/empty	Synchronization	Dataflow	System
Trans. memory	Clock	Reduce aborts	Speculative	Framework/library
Concurrent DS	Counter	Snapshot isolation	Lock-free	Data structure
Tasks	Task ID	Ordering	Dataflow	Runtime/compiler

Table I: Use-cases for versioning and renaming

by their identifiers. This ordering allows referring to the *latest* version in cases where the exact version needed is unknown. Programs must specify what version of the memory location they wish to load or store, and whether a load should be of the exact version or just capped by it. Loads directed to a version that is not yet created will block. All created versions are available simultaneously for loading. For instance, if versions 1 and 2 have both been created and stored to, then they can both be loaded from. Versions can also be created out of sequence. For example, version 2 may be stored to and loaded from before version 1 is created. Loads directed to version 1 would then stall, while those directed to version 2 would succeed — akin to register renaming in an out-of-order processor. We also introduce two new memory operations named LOAD-LOCK and UNLOCK, enabling ordering when dependencies cannot be statically analyzed in full.

O-structures support the following operations. For brevity, we omit the address from the list of O-structure operations, but in practice all operations take an address parameter (just like ordinary loads and stores).

- LOAD-VERSION: Given a version v, the call returns the value of that version. If the version has not been created yet or is locked, the call stalls. If another version of the same location is locked, then the lock is ignored and the value of the requested version is returned immediately.
- LOAD-LATEST: Given a version v, the call returns the value of the highest created version that is smaller than (or equal to) v. If no such version exists or it is locked, the call blocks.
- STORE-VERSION: Given a version v and a value, the call creates a new version and stores the value in it. Once created, a version can be locked but not modified.
- LOCK-LOAD-VERSION: Attempt a LOAD-VERSION and lock the loaded version if it succeeded. An attempt to lock an already locked version will stall.
- LOCK-LOAD-LATEST: Attempt a LOAD-LATEST and lock the loaded version if it succeeded. An attempt to lock an already locked version will stall.
- UNLOCK-VERSION: Given a previously locked version v_l and optional argument v_n, unlock v_l and optionally create a new version v_n with the same value as that stored in version v_l; v_n is left unlocked as well.

Use cases for the operations above will be presented in the next sub-section and demonstrated in Section IV. Briefly, LOAD-VERSION and STORE-VERSION allow fine-grain synchronization where dependence are known at compile time, e.g. on matrix multiplication. LOAD-LATEST allows some uncertainty, e.g. when a part of a dynamic data structure is modified by a single writer, causing each part of the data structure to contain different versions. Lastly, locking allows multiple concurrent writers to be synchronized when operating on some dynamic data structures.

Albeit optimized, O-structures should be used selectively, as they do incur latency and memory footprint overheads. Accesses to non-shared memory locations such as the stack do not benefit from using O-structures. Moreover, memory dedicated to I/O devices need not be in O-structures, as the semantics of I/O usually involves adhering to strict program order. Conventional (unversioned) and O-structure (versioned) functionality should be implemented at the microarchitecture level. Conventional memory is accessed using traditional LOAD and STORE operations while versioned memory uses the added instructions described above.

B. Programming model

O-structures interface is intentionally not coupled to a particular programming model. However, O-structures do provide specialized capabilities, which are a better match for certain use cases. Table I presents a non-exhaustive list of possible programming models that can utilize O-structures: Functional programming can use O-structures as I-structures [6], reducing versioning to full/empty bits, or as M-structures [7] utilizing renaming as well. Software transactional memory and concurrent data structures can use renaming to isolate readers from writers, whether in a speculative or non-speculative execution model. Lastly, parallel task-based execution of sequential code can use versioning to enforce ordering of true dependencies, renaming to eliminate false dependencies, and locking to provide ordering when dependencies are not known at compile time.

The low-level O-structures API is not intended to be used directly by programmers. Figure 1 lists two possible APIs, allowing parallelization of insertions into a linked list: (1) An extended compiler applies versioning and renaming based on programmer annotations, which specify the code structure that allows pipelining[1]; (2) A library intended for both programmers and compilers, wrapping the low-level API. The relation between the APIs resembles OpenMP and Posix threads. In our view, the higher-level API is preferable.

C. Practical considerations

O-structures interface and capabilities can be implemented purely as a software runtime abstraction; we've indeed started with a software prototype. However, the logic added to versioned memory operations incurred too much overhead, indicating hardware support is required. Software is still expected to provide the high-level programming model,

[1]The required compiler support is described in [8]; the pipelining technique is described in more details on Section IV and [9]

Unversioned code

```
struct node_t {
  node_t* next; ...
};

// assume non-empty list
node_t* root = init();

void insert_end(node_t* n) {
  node_t* prev = root;

  node_t* cur = root->next;
  while (cur != nullptr) {




    prev = cur;
    cur = cur->next;
  }
  prev->next = n;
}
// the outer loop
for (int i = 0; i < N; ++i)
  insert_end(new node_t{i});
```

Programmer annotations

```
struct node_t {
  [[versioned]] node_t* next; ...
};

[[pipeline_root]]
node_t* root = init();

[[task]] void insert_end(node_t* n) {
  node_t* prev = root;

  node_t* cur = root->next;
  while (cur != nullptr) {




    prev = cur;
    cur = cur->next;
  }
  prev->next = n;
}
[[pipeline]]
for (int i = 0; i < N; ++i)
  insert_end(new node_t{i});
```

Library versioning support

```
struct node_t {
  versioned<node_t*> next; ...
};
using vernode_t = versioned<node_t*>;

vernode_t root = init();

void insert_end(node_t* n, taskid_t tid) {
  vernode_t* prev = &root;
  // get a specific version of the head
  vernode_t* cur = root->next.lock_load_ver(tid);
  while (cur != nullptr) {
    // get latest ver. and block following task
    node_t* next = cur->next.lock_load_last(tid);
    // unlock prev. node and increment its version
    prev->next.unlock_ver(tid, tid + 1);
    prev = cur;
    cur = next;
  }
  prev->next.store_ver(n, tid);
}

for (int i = 0; i < N; ++i)
  create_task(i, insert_end, new node_t{i});
```

Figure 1: Parallelizing sequential insertions into the end of a linked list using two possible APIs: compiler support via C++11 attributes placed by the programmer, and library support wrapping low-level API.

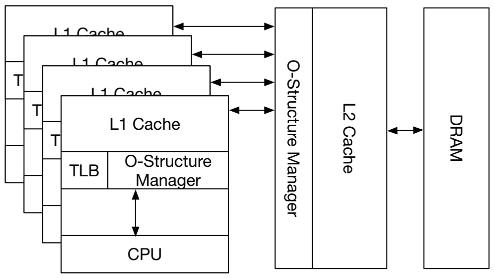

Figure 2: O-structures are primarily implemented as cache extensions. Processors use dedicated instructions to access them, but the rest of a system remains largely unchanged.

hence the architecture provides a clean, multi-purpose interface. Further, while dedicated instructions must be wrapped in a higher-level abstraction for reasonable programmer use, such an abstraction can be provided by a dedicated compiler.

Any practical implementation of O-structures needs to bound the number of live versions of data in the system. A poor solution is to temporarily switch from parallel to sequential execution.Then, only the latest version of each memory location will be needed, and all other versions can be safely discarded. This stop-the-world-like approach for garbage collection is not uncommon but is far from ideal. Instead, we describe a better approach, based on concurrent garbage collector ideas, in the next section.

III. Architectural Support

The O-structures microarchitecture is incorporated into a conventional cache hierarchy (Figure 2), and the processor is modified to dispatch the newly added instructions. Operating system involvement is also minimal, limited mainly to allocating memory regions that store versioned data. The remainder of this sections discusses the following top level design considerations: storage, addressing, caching and garbage collection. For brevity, we describe a 32-bit system; extending to 64-bit is trivial.

The microarchitecture supports two forms of version lookup: direct and full. In general, direct access is obtained by storing versioned data in the L1 cache, and augmenting the cache so versions are also considered in the lookup. When direct access fails, a full lookup is performed by traversing the list of versions. This might include accessing other cache levels, other cores or the RAM; we later describe optimizations mitigating lookup overheads.

Version Block: The core data structure used for O-structure storage is the *Version Block* (Figure 3). A version block is a 16 byte structure with five fields: a version identifier (32 bits), a pointer to the *physical address* of the next version block (30 bits), a locked-by field (32 bits), a bit indicating it is the head version block in a version block list (described below) and the actual data itself (32 bits[2]). The meta-data overhead for a single version of a program address is 12 bytes. We note that this design has overhead

[2]Versioning larger datums could be more efficient if fully utilized, but wasteful otherwise. We thus limit the initial design to the size of a pointer, and leave the evaluation of other sizes to future work.

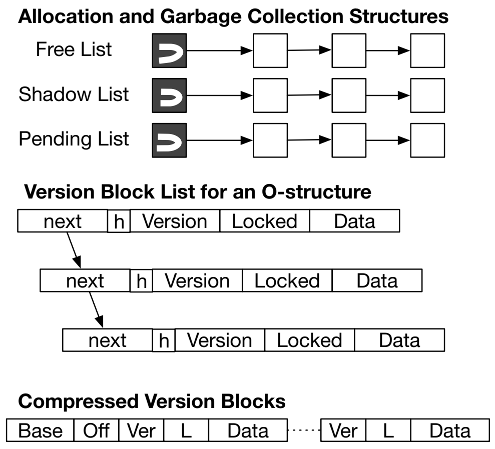
Allocation and Garbage Collection Structures
Free List
Shadow List
Pending List
Version Block List for an O-structure
next
h
Version
Locked
Data
Compressed Version Blocks
Base
Off
Ver
L
Data

Figure 3: The O-structure logic manages these data structures: free/shadowed blocks, version block lists, and compressed version block cache lines.

of 4× compared to an unversioned memory location. Below we describe how this overhead is reduced to 2× by storing version blocks in compressed form in the first level cache.

Version Block List: For a given address that contains an O-structure, each version of data is stored in its own version block. These blocks are linked together in a list, as depicted in Figure 3. The list is maintained in sorted order (newest in program order closer to the head), with version v_g closer to the head of the list than version v_l if $v_g > v_l$. The list is sorted in this way to simplify garbage collection of older versions (described later). Version Blocks are linked together into a list for a given program address. This list uses *physical* address pointers, as protection is provided on access to the head of the list (described below).

Free-list: Unused version blocks are stored in a free-list that is managed mostly by the hardware; OS handler is invoked only when available memory is exhausted. When a new version is created, a version block is pulled from the free-list and used as that version's storage. When the free-list is empty the hardware must notify the software, via a trap. version blocks are just ordinary memory structures, hence the runtime layer simply allocates more memory, carves it up into version blocks, and adds them to the free-list, modifying the page table as described below. Because of the physical pointers used to link together version blocks (even in the free list), extending the free-list must be a protected operation.

Addressing and protection: O-structures are referenced by user programs using *virtual* addresses. We adopt a protection scheme that extends the page table with a bit indicating that a page contains *version blocks*. Using conventional loads and stores to access version block pages generates a fault. Correspondingly, O-structure instructions fault if they reference pages whose *version block* bit is not set. Moreover, within a version block a field is provided to indicate that the block is the *head* of a version block list. The hardware checks this bit on an O-structure access; if the version block at the head of a list does not have this bit set, a fault occurs. This last check is necessary to prevent a user mode program from trying to access valid version blocks that are not at the head of a version block list. Note that while version blocks are linked using physical address pointers, restricting user programs to only access valid version blocks, and only via the head of version block lists, ensures that programs cannot use the versioning system to gain access to memory it would not otherwise have access to. Programs can corrupt their own memory space, but not the memory space of other programs or the system. Lastly, since block list lookups might require multiple memory accesses, the system also supports direct access (described below), thereby avoiding list traversal overhead.

A. *Caching*

O-structures can be supported by a system with conventional cache system, yet it is far more efficient to implement much of the O-structure logic in the cache system itself. In particular, data indexing and tagging should be extended to make optimal use of cache space. In this section we describe these changes, as they comprise the bulk of the microarchitectural implementation of O-structures.

Version lookup: To find a specific version of data in a given O-structure, the microarchitecture first attempts a *direct access*, relying on versioning support added to the caches. If the wanted version is already cached, its value will be immediately returned, incurring a latency similar to unversioned access to cached data. In case of a cache miss, a *full lookup* is performed by walking the version block list. The list is sorted, providing a mechanism to terminate early if a given version has not been created yet.

Data compression: To reduce versioning space overhead, version block data is stored in compressed form in the cache. version block compression relies on the observation that when more than one version of data exists for a given O-structure, the version numbers tend to be close (while this is not a requirement, versions are usually created in some order, rather than randomly generated). Close version numbers are found among lockers as well. For this reason we can compress eight version blocks into one 64 byte cache line as follows (Figure 3): we store an 18 bit *version base*, a 4 bit *cache line offset* and 8 compressed version block entries. Each compressed version block entry contains the data (32 bits), a 14 bit *version offset*, and a 14 bit *lock offset*. The cache line offset contains the offset within a 64 byte cache line for the head of the version block list, if cached. The version base contains the upper 18 bits of the lowest version stored within the cache-line. Version and locked-by are obtained by adding the version base field with the per-Version information. The only restriction thus imposed

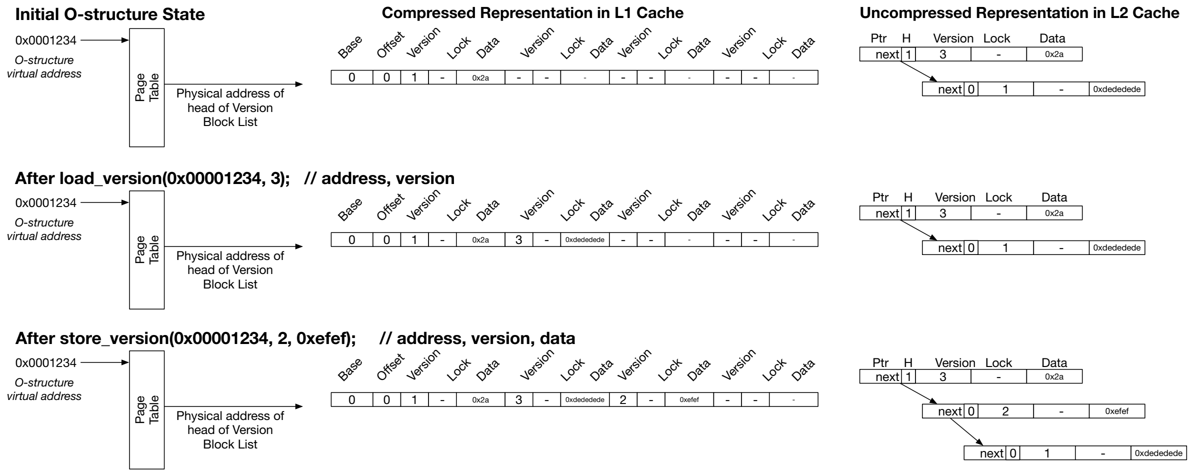
Initial O-structure State
0x0001234
O-structure virtual address
Page Table
Physical address of head of Version Block List
Compressed Representation in L1 Cache
Base
Offset
Version
Lock
Data
Uncompressed Representation in L2 Cache
Ptr
H
next
After load_version(0x00001234, 3); // address, version
After store_version(0x00001234, 2, 0xefef); // address, version, data
0x2a
0xdededede
0xefef

Figure 4: Access to and storage of O-structures in memory. O-structure accesses first go through two indirections in the page table, preventing user mode programs from fabricating O-structure addresses and permitting fine-grained interleaving of O-structure user addresses with conventional program locations. O-structures are stored in compressed form in the L1 cache and uncompressed form in the L2 cache.

by the compression is on the range of versions and lockers within a single cache line.

Compressed version block information is stored in the cache with only a 2× overhead. But more importantly, compressed version block data is accessed in a single cache lookup. To perform a direct access, the cache accesses the line with the physical address of the head of the version block list. If that cache line is stored in compressed form, indicated by a bit in the meta-data of each cache line, the cache-line data is inspected to search for a match. If no match is found (or if the cache line is not stored in compressed form), then a direct access falls back to a full lookup. Caches can choose any appropriate (e.g. LRU) policy to manage versions within a compressed cache line.

Supporting the simultaneous mixing of direct and full lookups in the cache is straightforward. Caches that are at least two-way associative can store both compressed and uncompressed versions of an O-structure at the same time; direct mapped caches can use multiple hash functions to achieve a similar functionality. Compressed O-structures do require a slight change to the coherence protocol: messages for a cache line that contains a version block must also convey the physical address of the head of the version block list. Since modifications to a version block always begin by obtaining the physical address of the version block list, the address is always readily available prior to sending any coherence message. When caches receive a coherence message, the simplest course of action is to discard the compressed version block for that O-structure. Subsequent requests to access version data can rebuild the compressed version block. Sophisticated approaches that modify compressed version blocks in situ are left for future work.

Avoiding cache pollution: When a processor requests a specific version, the cache may end up traversing a fairly long version block list. To avoid cache pollution, only the block that holds the requested version is inserted into the cache; other blocks that were fetched during traversal are not cached. Since direct version accesses outnumber traversals, cold versions thus do not take the place of hot ones.

Adding a new version: Adding a new version of data to an O-structure requires that a version block be removed from the free-list. The version block list is traversed to locate the version block (or root pointer) that must be modified to point to this new version block. To avoid dead-lock, the two cache lines that must be modified – the one containing the version block already in the list and the one containing the new version block that is to be inserted into the list – must be acquired for exclusive access in a well-defined order (e.g. lowest address first). Deciding where in the list to insert a new version block and acquiring exclusive access to the necessary cache lines might cause a race. Consequently, the version block already in the list must be re-checked to ensure that it has not been modified. If it has, the procedure must be retried. After the new version block is inserted into the list, exclusive access to both cache lines is no longer required.

Locking a version: Locking a version requires gaining exclusive access to the version block that contains that version and modifying the lock field appropriately. Once the lock field is modified, exclusive access is no longer required until the version is to be unlocked. However, an extended coherence protocol can maintain the exclusive access rights until the block is unlocked, since no other task should be able to read or lock that version anyway.

Summary: Figure 4 depicts operations and structures described above. First, it shows a compressed L1 line holding only the head of a Version Block list, and the uncompressed list on L2. After a LOAD-VERSION, another version is placed in the compressed line. Finally, a new version is inserted in between existing versions on the Version Block list and at a vacant slot in the compressed line.

B. Garbage collection

O-structures are most suitable for use in task-based execution models that parallelize sequential code[3]. When task IDs reflect the sequential order of the program, mapping task IDs to versions naturally orders versions and dependencies. This ordering allows unneeded versions to be identified and safely discarded. A version becomes unneeded once it is *shadowed* by a younger version, and no active (or future) task will try to lock or load it. The following section describes a garbage collection mechanism, which allows automatic, efficient, on-the-fly management of version blocks. The garbage collector is implemented in hardware, and relies on progress reports provided by the program. Further, the run-time system must follow a set of simple rules to ensure safe garbage collection.

Program cooperation: The versioning instruction set is decoupled from the execution model. This makes the design more generic but hinders garbage collection. To provide the memory system with enough information for safe version reclamations, the garbage collector expects the runtime to fulfill a small set of rules. Those rules are described below in terms of parallel task-based execution or sequential code, but as said, can be adapted to other models:

1) Use the task ID when accessing versions, whether for read or for write, ensuring that the ordering of versions matches the ordering of the sequential program.
2) Inform the memory system when a task begins and ends, allowing progress to be monitored. This is done using two dedicated new instruction, TASK-BEGIN and TASK-END.
3) Ensure that a task with an ID *lower* than the ID of the lowest active task in the system is not created. This merely places a lower bound on the IDs of active and future tasks (out-of-order spawning is still permitted).

These restrictions are easy to satisfy and allow unneeded versions to be identified and reclaimed.

Version shadowing: The garbage collection process relies on the concept of *version shadowing*. Once a new version of some location is created, the old version is shadowed; and once no task may read the old version, it can be reclaimed. Figure 5 depicts the following scenario: task 1 creates version 1, which task 2 may read. Task 3 then creates version 3, which shadows version 1 for future tasks. Once task 2 ends, no task will access version 1, and it can be reclaimed. In general, rule #3 ensures that at some point there will be no active task with an ID smaller than the

[3]Adapting garbage collection to other models is omitted for brevity.

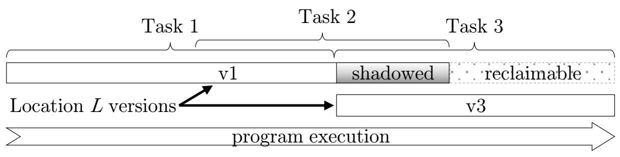

Figure 5: When task 3 creates version 3 of location L, it shadows version 1 from future tasks. Once task 2 ends, version 1 becomes reclaimable.

last possible reader of a shadowed version. From that point onward, rule #1 guarantees that no task will try to access that version, as all active tasks will access newer versions. From rule #2, the garbage collector can monitor the range of active tasks, and occasionally reclaim such versions.

Operation: The garbage collector manages two lists: (1) the *Shadowed list* holds a list of shadowed version blocks. The blocks may still be accessed by the program, but will become eligible for reclamation at some future time; (2) the *Pending list* holds a list of shadowed blocks, obtained from the shadowed list. Those blocks can be reclaimed at the end of the garbage collection phase.

When a new version is created, if it shadows a previous version, then that shadowed version will be registered in the shadowed list. Namely, the shadowed list holds versions of various memory locations that *might* be inaccessible to the program. When a garbage collection phase is initialized, the shadowed list is moved to the pending list, and the youngest active task is recorded. Once the oldest active task is younger than the recorded task, the garbage collection phase is finalized, moving everything from the pending list to the free list. Throughout the collection phase, newly shadowed versions are added to the shadowed list as usual.

The safety of the garbage collection algorithm is ensured by tracking the program's execution state and discarding versions that are not reachable by any task. The pending list allows an automatic, on-the-fly collection. Without an intermediate list, execution would have to be serialized and stopped, to ensure that all shadowed versions are no longer reachable by any task. In general, the garbage collection phase can be invoked by hardware or software. Our solution is hardware-based, and automatically invokes the collection when the number of free blocks drops below a watermark.

C. Allocating and Freeing O-structures

The final challenge regarding memory management is the reuse of a versioned address, which holds a pointer to the version block list. An ordinary memory address is converted to an O-structure simply by ensuring that its value is null and starting to perform versioned accesses to it. Converting an address from an O-structure to a conventional memory location requires that no unfinished task access that location as an O-structure. Fortunately, the protection model

Parameter	Value
Processor	2-way in-order (ARM ISA), 2GHz
L1 I/D Cache	32KB, 8-way associative, 64B block, 4 cycles hit latency
L2 Cache	$1.5MB \times \#cores$, shared, 16-way associative, 64B block, 35 cycles hit latency
Memory	64GB, 60ns latency

Table II: The experimental platform

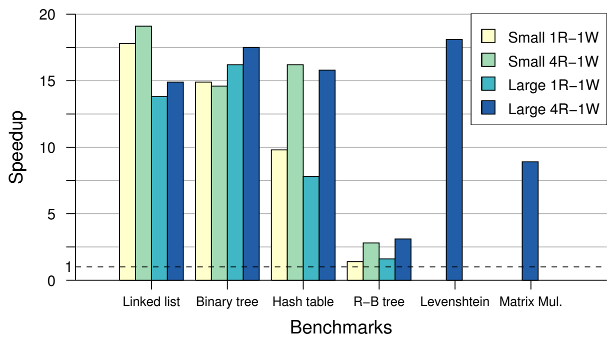

Figure 6: Speedup of parallel versioned (32 cores) over sequential unversioned. Small benchmarks start with 1000 elements and large with 10000; Read-intensive (4R-1W) is 4 reads per write and Write-intensive (1R-1W) is 1 read per write.

ensures that violations of these rules will only cause errant behavior for the process, and cannot corrupt other processes or the operating system. Nevertheless, these restrictions do warrant caution for dynamic memory allocation. The simplest solution is for programs to delay the recycling of memory after it is freed until points of execution where no parallel tasks are executing. Another technique is to merge the garbage collection of version blocks with the runtime system's malloc pool. This is a subject of future work.

IV. Evaluation

In this section, we evaluate O-structures implementation and the benefits they provide. We begin with matrix multiplication and Levenshtein distance, which require ordering based on versioning, but do not require renaming and locking. We then evaluate a concurrent data structure, and show how versioning provides snapshot-isolation for readers. Finally, we evaluate a set of irregular data structures supporting concurrent readers and writers, and show how O-structure's fine-grain locking allows deterministic results.

A. Methodology

We implemented an O-structure memory system in the gem5 [10] simulator executing in system emulation (SE) mode. The system configuration is detailed in Table II. The task scheduler was implemented in software and used a static

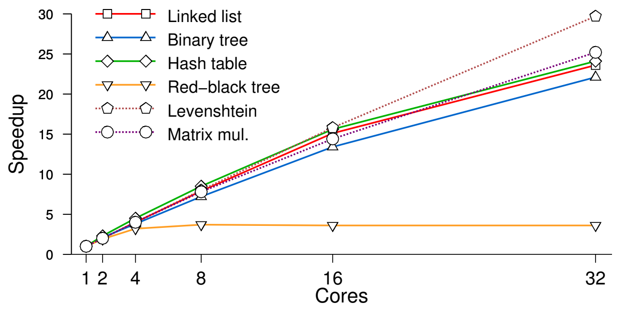

Figure 7: Scalability analysis (speedup over sequential versioned; large, read-intensive runs).

assignment of tasks to cores. This policy imposes a minimal runtime overhead, but neglects load imbalance.

The regular data structures evaluation included two workloads: multiplication of 3 dense 100*100 matrices[4] and Levenshtien distance between strings of size 1000. The implementation is a direct translation of the sequential code, augmented with versioning to allow parallel execution.

The snapshot isolation evaluation compared an unversioned binary tree using a read-write lock and a similar tree using versioning and renaming. Workloads were read intensive: 3 reads per 1 write; initial tree size was 10000. Scan ranges of 1, 8 and 64 were evaluated.

The task-based execution evaluation considered three aspects of the implementation: (1) *Memory footprint:* the initial number of elements was *small* (1000) or *large* (10000), yielding different effective memory footprints and accordingly different caching efficiency; (2) *Operations ratio:* read-only operations (lookups) are more common than mutating operations (inserts and deletes). We evaluated two read-write ratios – *read-intensive* (4 reads per write) and *write-intensive* (1 read per write); and (3) *versioned vs. unversioned:* self-speedup of versioned code indicates scalability, but versioning adds overhead. The real benefit is obtained by comparing parallel versioned to sequential unversioned code.

B. Versioning only — regular data structures

Matrix multiplication and Levenshtein distance are mostly data parallel algorithms and scale very well. Neither algorithm requires renaming since each memory location is written to only once, and they both use O-structures as I-structures [6]. Figure 6 shows versioning can add a non-trivial overhead on a *single-threaded* execution, e.g., matrix multiplication runs 2.5× slower than a conventional unversioned implementation due to the very large number of O-structures and versioned operations. However, the overhead is fixed, and Figure 7 shows that a multi-threaded

[4]Due to the complexity of the algorithm, larger (and more realistic) workloads could not be simulated in reasonable time.

execution scales well (comparing parallel execution to a single-threaded versioned run). The key takeaway is that although their full set of features might add overhead even if not fully utilized, O-structures can be effectively used to parallelize regular algorithms.

C. Versioning and renaming — snapshot isolation

Concurrent data structures often guarantee snapshot isolation. This ensures that scans are serializable, i.e., their interaction with concurrent updates does not result in a set of results that would not exist if the updates had been executed sequentially. A common way to obtain such isolation is by versioning and renaming, effectively eliminating write-after-read dependencies. Another common strategy is to separate reads and writes, eliminating synchronizations but also concurrency.

Figure 8 presents a comparison of O-structure-based binary tree with an unversioned binary tree, protected by a read-write lock [5] . Snapshot isolation is obtained by renaming and separating writes and reads, respectively. Both execute the same series of inserts and scans, and 3 scan ranges are considered. The unversioned code runs faster on a single-threaded execution due to the versioning overhead. However, on a parallel execution, the unversioned implementation executes reads and writes separately, yielding an average self-speedup of 7.9×. On the versioned implementation, inserts and scans can overlap, yielding a self-speedup of 12.2×. The higher scalability allows the versioned implementation to outperform the unversioned one by 16% on average. While this is not a huge win, snapshot isolation is just one of the use cases for the proposed architectural facility.

D. Versioning, renaming and locking – task-based execution

The last group of benchmarks uses a parallel task-based execution of sequential code. Namely, sequential code is divided to tasks, which are executed in parallel. The benchmarks operate on widespread irregular data structures in which dependencies are not known in advance. Ordering is maintained by using the following algorithm (simplified):

- The root of the data structure is entered in-order, relying on LOCK-LOAD-VERSION (mutating tasks) or LOAD-VERSION (read-only tasks)
- During traversal, hand-over-hand locking using LOCK-LOAD-LATEST maintains ordering of dependent tasks following the same path
- Modifying pointers renames them using STORE-VERSION to eliminate anti-dependencies

[5]In terms of programming effort, using a read-write lock is similar to using O-structure versioning. More involved algorithms (e.g. lock-free) supporting snapshot isolation do exist, but are much harder to implement. Further, our experience with implementing O-structures in software shows they perform much worse than hardware-based O-structures; therefore, we do not use software renaming as a baseline.

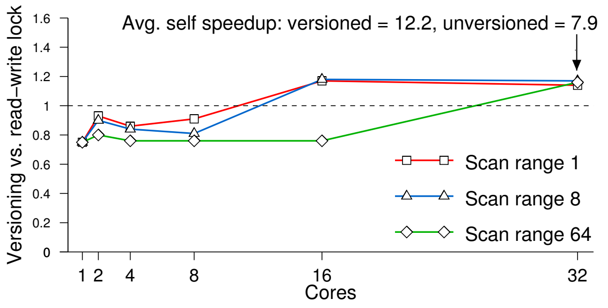

Figure 8: Performance ratio between versioned binary tree and unversioned tree protected by a read-write lock. Initial tree size is 10,000, measured operations include scans and inserts in a 3:1 ratio. Scan ranges are 1 (simple get), 8 and 64. Above 1 means versioned implementation runs faster.

The result is fine-grained pipelining of operations, requiring the minimal amount of RaW synchronization. The output of such parallel execution is identical to a sequential execution.

Our evaluation of irregular data structures interleaves lookups, inserts and deletes in different ratios on pre-populated data structures. The number of insertions and deletions was set to be equal, so the effective memory footprint does not change significantly during the run. The per-task amount of work was also held steady due to the stable size of the data structures. The speedups comparing to sequential unversioned runs are presented in Figure 6, and scalability of versioned executions in Figure 7. The key findings from the evaluation are as follows:

Inherently poor locality masks the overhead of cross-core communication. On the linked list benchmark, all runs experience very low L1 hit rate (below 50%). Other data structures have a higher hit rate, but pointer-based data structures have inherently poor locality. However, as opposed to stalls due to ordering, capacity misses can be handled in parallel, and do not extend the critical path. Further, obtaining cache lines from the LLC or another core have comparable latencies, hence scaling out does not incur high communication penalty.

Root ordering can form a bottleneck. The dependency-safe traversal relies on ordering done at the root. This ordering forms a bottleneck, whose effect depends on the rate in which tasks enter the data structure. On write-intensive hash tables, up to 85% of versioned root loads are stalled. However, readers do not lock the root, hence read-oriented hash table workloads have a much lower stall rate. Binary trees have a low depth to size ratio, hence also experience stalls (40% on small trees, 25% on large ones).

For sufficiently large traversals, true memory dependencies do not limit parallelism. Linked lists experience

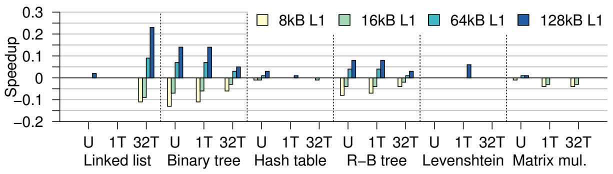

Figure 9: The performance impact of changing the L1 cache size between 8kB–128kB compared to a 32kB baseline; unversioned (U), versioned single core (1T) and versioned 32 cores (32T) runs are presented. The figure demonstrates that increasing the L1 cache size beyond 32kB has limited impact — up to $1.23\times$ and usually much less.

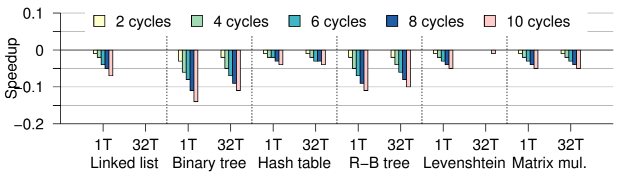

Figure 10: The negative speedup (slowdown) caused by injecting 2–10 cycle latencies into versioned ops vs. no versioning overhead; versioned single core (1T) and 32 cores (32T) are shown.

very few collisions that force versioned loads to stall while the target version is locked, since traversals are long. Trees and hash tables provide shorter pipelines, but diverge fast once the root is passed. The use of fine-grained locking thus ensures that tasks accessing different locations will never synchronize. Interestingly, on 32-core linked list runs, mutating tasks moving slowly rarely stall (15% of versioned loads), yet read-only tasks moving fast are frequently stalled trying to bypass older mutators.

Algorithmic optimizations are sometimes required to increase speedup. The red-black tree benchmark is an attempt to handle balanced data structures, which are harder to parallelize due to the rebalancing procedure. Our implementation allows a single writer, and readers might see a slightly unbalanced tree. This severely limits parallelism, forcing the root to heavily throttle traversals. While task pipelining is limited in red-black trees, particular parts can be optimized. For instance, in our baseline the delete operation was locking a deleted pointer longer than necessary; algorithmic modifications shortened the locking.

E. Size and latency sensitivity

We complement our gem5 model by examining the performance impact of various L1 sizes and O-structure operation latencies. The analysis shows that increased memory footprint and higher cache latency have little effect on speedups.

L1 size: Even if the proposed compression scheme is fully utilized, O-structure storage does increase the memory footprint. To better understand the effect of L1 size, we ran all large, read-intensive benchmarks using a range of simulated L1 sizes. The results, presented in Figure 9, indicate that L1 size does not significantly affect unversioned, sequential versioned, or parallel versioned runs. For instance, let us examine the binary tree numbers: on a 8kB L1, the read miss rate of unversioned run is 22%. Increasing L1 capacity to 128kB lowers the miss rate to 9%, which translates to a speedup of $1.3\times$. The sequential versioned run has slightly higher miss rates (25% and 11%) and a similar speedup. The miss rates observed when running on 32 cores are higher due to data sharing (28% and 21%), which reduces performance by 10%. This goes to show that while L1 size has a noticeable effect on sequential runs, the effect is much smaller than that of parallelizing the code; on parallel runs, the L1 size matters less. Admittedly, data structures that fully fit in L1 might see little gain from parallelizing. However, they are usually not a performance concern anyway. Further, loads make about 25% of instructions on average; the rest will benefit from parallelization regardless of L1 size.

L1 latency: Support for O-structures mostly relies on extending the L1 cache logic. While most of the versioning logic can execute in parallel with other operations (e.g., comparing versions and tags), determining the exact latencies of versioned operations requires a detailed RTL model. Instead, we estimate the influence of L1 latencies by injecting a fixed latency to every versioned operation. Figure 10 shows that adding 10 cycles to each versioned access reduces performance by up to 16%. The impact is much milder when using smaller (and more realistic) latencies. These findings correlate with the L1 size evaluation, which indicates a fairly high miss rate: frequently accessing the LLC reduces the effect of L1 latency.

Irregular data structures generally make suboptimal use of some central micro-architectural mechanisms: the large memory footprint overflows lower caches; irregular layout forbids prefetching; and data-dependent conditional branches yield a high rate of misprediction [11]. These issues are orthogonal to O-structures – any mechanism designed to mitigate them would also work with O-structures. For the results presented in this paper, however, the inherently low IPC of irregular codes explains why the results are relatively insensitive to cache size and latency.

F. Garbage collection overhead

Our Memory Version Manager is designed to perform garbage collection in the background, adding very little latency to the execution's critical path. To evaluate its overhead, we executed a sequential workload of 1000 operations on a sorted linked list with 10 elements. The small list size magnifies the effect of version allocation. A tight configuration that triggered 135 garbage collection phases was only 0.1% slower than a configuration with enough free versions to avoid ever executing the garbage collection process. Further, the latter configuration was also 0.1% slower than a configuration with no version sorting. While this test only considered versions that were created in-order (hence sorted), that is the common case in real programs.

V. Related Work

A. Dataflow memory

Dataflow machines have a unique set of memory challenges created by the lack of a total store load order. This has been addressed using a variety of creations. Arvind et al. introduced I-structures [6], which are write-once memory locations. Barth et al. [7] later extended I-structures for mutable computation with M-structures, which are memory locations with full/empty bits that can be written to ("filled") and read from ("emptied"). I-structures and M-structures, however, do not provide total ordering between an arbitrary number of producers and consumers. Specifically, since I-structures and M-structures do not provide versioning, they can only serve as rendezvous points between exactly one producer and its consumers. Consequently, the ordering of producers and consumers is lost when executing multiple instances of a task. Indeed, the Cray MTA/XMT series of machines [12] used an M-structure-like concept of full/empty bits [13] for thread synchronization. This model enables programs to track trivial data dependencies in memory. Other studies extended this model to data versioning without renaming [14] or limited dependency tracking with data renaming using a coarse-grain memory buffer level, eliminating a subset of false dependencies [2].

Other dataflow machines such as WaveScalar [15] statically encoded the program control flow graph on the memory operations themselves so that total store load order can be reconstructed from dataflow execution. In contrast, O-structures *dynamically* enforce a total store order, which facilitates out-of-order memory parallelism.

B. Implicit memory versioning

The concept of implicitly storing multiple versions of a memory element for synchronization and checkpointing is common in *transactional memory* (TM) and *thread-level speculation* (TLS). This paper demonstrates that a hardware-based memory versioning can be scaled to efficiently support various programming models.

Transactional memory [16] implementations use versioning for two purposes: speculation and snapshot isolation. TM enables multiple threads to speculatively produce a new versions for a collection of data elements and, ultimately, commit at most one thread's version. TM proposals use either the cache system to store speculative versions (e.g., [16], [17], [18]) or the memory itself (e.g., [19], [20]). Storing multiple non-speculative versions provides snapshot isolation, allowing read-only transactions to always complete without aborting. This has been used in both software [3], [21] and hardware [19] implementations.

Thread-level speculation is a technique in which multiple threads operate in parallel, and mutations of each thread are committed to the shared state only if consistent with respect to the previous state. While the use of coarse-grained speculation resembles that of TM, TLS targets sequential code. As opposed to TM, in which transaction boundaries are specified by the programmer, TLS synchronizes tasks, which are obtained from the program by the compiler or the hardware, and the sequential ordering drives commit order.

TLS has yielded many implementations (e.g., [22], [23], [24], [25], [26], [27], [28], [29], [30], [31]). MultiScalar [22], which arguably spearheaded research on TLS architectures, is a task-based parallel architecture designed to execute imperative language programs. MultiScalar relies on hardware prediction techniques [32] for aliases, and an innovative structure called Speculative Versioning Cache (SVC) [33] to scale past the limits of centralized store-queue designs.

TLS and O-structures-based tasks both target sequential programs and use versioning as part of the implementation, but both the programming model and execution strategy differ significantly. First, TLS is not exposed to the programmer or the program. This makes TLS easier to adopt, but leaves no way for the programmer or compiler to express their knowledge of the program and the available parallelism. By leveraging that knowledge, task-based programming models using O-structures can benefit from larger amounts of *non-speculative* parallelism. Second, relying on speculations allows TLS to be used with a wide range of code patterns, while the use of O-structures relies on known dependencies or patterns such as pipelining. However, while TLS implementations such as SVC are limited in the number of the data dependencies they can track, O-structures can track versioning at memory-scale, eliminating task size limit.

C. Loosely related mechanisms

The need to track fine-grain data dependencies has motivated research on hardware support for explicit parallel programming. One notable direction targets low-latency communication across threads and tasks. These systems include mechanisms such as scalar operand networks [34], direct register-level communication [35], [36], or a shared register file [37]. Nevertheless, these studies primarily address register-level dependencies, whereas O-structures target memory-level dependencies at scale.

VI. Conclusions

This paper introduced architectural support for unlimited memory versioning, renaming and fine-grained locking, using O-structures. The extended memory interface can be leveraged by various programming models, frameworks and libraries, using a subset or all of O-structures features. To demonstrate the feasibility of the proposed mechanism, the paper also describes a supporting instruction set architecture and a microarchitectural implementation that maintains desirable system properties such as process isolation and bounded memory consumption.

While O-structures can be implemented entirely in software, microarchitectural support is necessary to see significant performance gains compared to sequential execution. Fortunately, microarchitectural support is confined mostly to the cache system and a few additional instructions. Our simulation-based study of this microarchitecture suggests that significant speedup of up to 19× is achievable on pointer-heavy benchmarks, which utilize the complete set of O-structure features.

ACKNOWLEDGEMENTS

This research was funded, in part, by Google and the Israel Ministry of Science, Technology, and Space. Eran Gilad and Elazar Raab were supported by fellowships from the Hasso-Plattner Institute.

REFERENCES

[1] M. Franklin and G. S. Sohi, "ARB: A hardware mechanism for dynamic reordering of memory references," *IEEE Trans. on Computers (TC)*, vol. 45, no. 5, May 1996.

[2] Y. Etsion, F. Cabarcas, A. Rico, A. Ramirez, R. M. Badia, E. Ayguade, J. Labarta, and M. Valero, "Task superscalar: An out-of-order task pipeline," in *Intl. Symp. on Microarchitecture (MICRO)*, 2010.

[3] T. Riegel, C. Fetzer, and P. Felber, "Snapshot isolation for software transactional memory," in *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, 2006.

[4] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer, "An asymptotically optimal multiversion b-tree," *Very Large Data Bases Journal (VLDBJ)*, vol. 5, no. 4, Dec. 1996.

[5] E. Gilad, E. W. Mackay, M. Oskin, and Y. Etsion, "O-structures: Semantics for versioned memory," in *Workshop on Memory Systems Performance and Correctness (MSPC)*, 2014.

[6] Arvind, R. S. Nikhil, and K. K. Pingali, "I-structures: data structures for parallel computing," *ACM Trans. on Programming Languages and Systems (TOPLAS)*, vol. 11, no. 4, Oct. 1989.

[7] P. S. Barth, R. S. Nikhil, and Arvind, "M-Structures: Extending a parallel, non-strict, functional language with state," in *Conference on Functional Programming Languages and Computer Architecture (FPCA)*, 1991.

[8] T. Mayzels, "Software management of hardware memory versioning," Master's thesis, Technion, 2017.

[9] E. Gilad, T. Mayzels, E. Raab, M. Oskin, and Y. Etsion, "Towards a deterministic fine-grained task ordering using multi-versioned memory," in *Computer Architecture and High Performance Computing (SBAC-PAD)*, 2017.

[10] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *Computer Architecture News (CAN)*, vol. 39, no. 2, 2011.

[11] O. Green, M. Dukhan, and R. Vuduc, "Branch-avoiding graph algorithms," in *Symp. on Parallel Alg. and Arch. (SPAA)*, 2015.

[12] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, "The Tera Computer System," in *ACM Intl. Conf. on Supercomputing (ICS)*, 1990.

[13] B. Smith, "Architecture and applications of the HEP multiprocessor computer system," in *Real-Time signal processing (RTSP)*, 1981.

[14] C. Segulja and T. Abdelrahman, "Architectural support for synchronization-free deterministic parallel programming," in *Symp. on High-Performance Computer Architecture (HPCA)*, 2012.

[15] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "WaveScalar," in *Intl. Symp. on Microarchitecture (MICRO)*, 2003.

[16] M. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," in *Intl. Symp. on Computer Architecture (ISCA)*, 1993.

[17] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional memory coherence and consistency," in *Intl. Symp. on Computer Architecture (ISCA)*, 2004.

[18] R. Rajwar, M. Herlihy, and K. Lai, "Virtualizing transactional memory," in *Intl. Symp. on Computer Architecture (ISCA)*, 2005.

[19] H. Litz, D. Cheriton, A. Firoozshahian, O. Azizi, and J. P. Stevenson, "SI-TM: Reducing transactional memory abort rates through snapshot isolation," in *Arch. Support for Programming Languages & Operating Systems (ASPLOS)*, 2014.

[20] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood, "LogTM: log-based transactional memory," in *Symp. on High-Performance Computer Architecture (HPCA)*, 2006.

[21] D. Perelman, A. Byshevsky, O. Litmanovich, and I. Keidar, "Smv: selective multi-versioning stm," in *International Symposium on Distributed Computing (DISC)*, 2011.

[22] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar processors," in *Intl. Symp. on Computer Architecture (ISCA)*, 1995.

[23] J. Oplinger, D. Heine, S.-W. Liao, B. A. Nayfeh, M. S. Lam, and K. Olukotun, "Software and hardware for exploiting speculative parallelism with a multiprocessor," Stanford Univ., Tech. Rep., 1997.

[24] H. Akkary and M. A. Driscoll, "A dynamic multithreading processor," in *Intl. Symp. on Microarchitecture (MICRO)*, 1998.

[25] V. Krishnan and J. Torrellas, "A chip-multiprocessor architecture with speculative multithreading," *IEEE Trans. on Computers (TC)*, vol. 48, no. 9, Sep 1999.

[26] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry, "A scalable approach to thread-level speculation," in *Intl. Symp. on Computer Architecture (ISCA)*, 2000.

[27] M. Cintra, J. F. Martinez, and J. Torrellas, "Architectural support for scalable speculative parallelization in shared-memory multiprocessors," in *Intl. Symp. on Computer Architecture (ISCA)*, 2000.

[28] J. Renau, J. Tuck, W. Liu, L. Ceze, K. Strauss, and J. Torrellas, "Tasking with out-of-order spawn in TLS chip multiprocessors: Microarchitecture and compilation," in *ACM Intl. Conf. on Supercomputing (ICS)*, 2005.

[29] C. Madriles, C. García-Quiñones, J. Sánchez, P. Marcuello, A. González, D. M. Tullsen, H. Wang, and J. P. Shen, "Mitosis: A speculative multithreaded processor based on precomputation slices," *IEEE Trans. on Parallel and Distributed Systems (TPDS)*, vol. 19, no. 7, Jul 2008.

[30] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry, "The STAMPede approach to thread-level speculation," *ACM Trans. on Computer Systems (TOCS)*, vol. 23, no. 3, Aug. 2005.

[31] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez, "A scalable architecture for ordered parallelism," in *Intl. Symp. on Microarchitecture (MICRO)*, 2015.

[32] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi, "Dynamic speculation and synchronization of data dependences," in *Intl. Symp. on Computer Architecture (ISCA)*, 1997.

[33] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi, "Speculative versioning cache," in *Symp. on High-Performance Computer Architecture (HPCA)*, 1998.

[34] M. Taylor, W. Lee, S. Amarasinghe, and A. Agarwal, "Scalar operand networks: On-chip interconnect for ILP in partitioned architectures," in *Symp. on High-Performance Computer Architecture (HPCA)*, 2003.

[35] S. Campanoni, K. Brownell, S. Kanev, T. M. Jones, G.-Y. Wei, and D. Brooks, "HELIX-RC: An architecture-compiler co-design for automatic parallelization of irregular programs," in *Intl. Symp. on Computer Architecture (ISCA)*, 2014.

[36] S. Srinath, B. Ilbeyi, M. Tan, G. Liu, Z. Zhang, and C. Batten, "Architectural specialization for inter-iteration loop dependence patterns," in *Intl. Symp. on Microarchitecture (MICRO)*, 2014.

[37] F. Karim, A. Mellan, A. Nguyen, U. Aydonat, and T. Abdelrahman, "A multilevel computing architecture for embedded multimedia applications," *IEEE Micro*, vol. 24, no. 3, May-June 2004.