Towards a Deterministic Fine-grained Task Ordering using Multi-Versioned Memory

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

***Abstract*—Task-based programming models aim to simplify parallel programming. A runtime system schedules tasks to execute on cores. An essential component of this runtime is to track and manage dependencies between tasks. A typical approach is to rely on programmers to annotate tasks and data structures, essentially manually specifying the input and output of each task. As such, dependencies are associated with named program objects, making this approach problematic for pointer-based data structures. Furthermore, because the runtime system must track these dependencies, for efficient runtime performance the read and write sets should be kept small.**

We presume a memory system with architecturally visible support for multiple versions of data stored at the same program address. This paper proposes and evaluates a task-based execution model that uses this versioned memory system to deterministically parallelize sequential code. We have built a task-based runtime layer that uses this type of memory system for dependence tracking. We demonstrate the advantages of the proposed model by parallelizing pointer-heavy code, obtaining speedup of up to $19\times$ on a 32-core system.

I. Introduction

Task-based programming models present the developer with an abstract parallel machine. Typically this abstract machine has useful guarantees for the developer, such as an order in which the tasks will be completed, or when parallel execution will be synchronized. Furthermore, the abstract machine is designed to hide many low-level details such as which cores execute which tasks and in which order.

Many task-based programming models rely on dependency and output lists, specified by the programmer, to provide non-speculative and thread-safe parallel execution (e.g., [1], CellSs [2], TaskSs [3], OpenMP 4.0 [4] and OpenStream [5]). Input and output specifications are conveyed to the runtime and stored as part of each task's metadata. Task execution is then driven by the dataflow *firing rule* [6], which dictates that a task is considered ready for execution only once all of its inputs have been produced by other tasks. Relying on the firing rule ensures that once a task is ready, it can be executed from start to end without worrying about dependencies.

A major shortcoming of the firing rule is the expressiveness of the dependency specification. In order to mark a particular datum as a dependency, that datum must be *named* and the task-based runtime notified. Naming can rely on a scalar variable, single pointer dereference or array offset. Some models support the naming of high-level data-structures with associated semantics, such as streams in OpenStream. Notably, all of these naming schemes exclude internal nodes of pointer-based data structures from being specified as a dependency — such nodes are *anonymous* even if the node within the data structure is pointed to by some variable. Consequently, the use of pointer-based data structures in task-based programming models is restricted: typically the *entire* data-structure is treated as a single object, leading to excessive synchronization between tasks.

Dependency specification is not a problem for an alternative task-based execution strategy: speculation. In the speculative execution model, tasks can be executed even if their dependencies are unknown in advance. Once tasks complete, they must be committed. During the commit phase, dependencies are checked; if a violation is identified, the task is re-executed.

Since speculative task execution does not rely on dependency specification and the firing rule, it can cope with anonymous dependencies. Consequently, dynamic data structures can be used for shared state and be accessed by multiple tasks. However, speculative execution introduces another limitation: read and write sets. Those are used when checking dependencies on the commit phase. Managing and checking the sets in software allows unlimited size but is slow. Hardware-based solutions are faster but are typically limited in set sizes.

This paper proposes the Multi-Versioned Memory (MVM) model, a task-based programming and execution model that relies on O-structures [7], an extended memory system providing data versioning and renaming. Versioning is used to embed dependencies in the memory system, eliminating the need for a predefined dependency list. In the absence of dependency specification as part of the tasks metadata, the model cannot rely on the firing rule for scheduling. Instead, tasks are considered ready as soon as they are created, and can be launched according to any scheduling policy. During execution, if a task tries to access a version that does not exist, the memory system stalls it until the version is created. The result is a fine-grain, deterministic, dataflow execution model. In addition, data renaming is obtained by storing multiple versions of the same datum. This eliminates false dependencies, further increasing parallelism and reducing the need to track accesses to shared locations.

Execution under the MVM model is designed to have the *statically sequential* property [1]. Namely, the parallel execution of a program should yield identical results to a sequential execution of the same program. This is done by mapping task identifiers to their sequential ordering. Task identifiers are then used for versioning, effectively reflecting the sequential ordering of the program in the memory system.

The interface and semantics of the memory system required

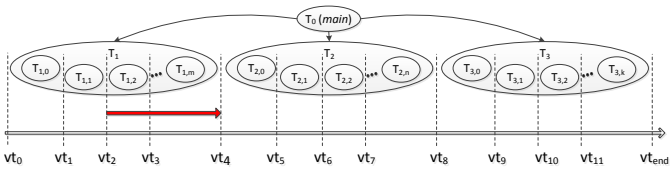

Fig. 1: Mapping of a hierarchical, task-based decomposed program onto a monotonic virtual timeline (horizontal axis). The decomposition enforces the sequential order of inter-task memory dependencies. This is known as the *statically sequential* property of task-based programming models [1].

for MVM are an extension of O-structures [7]. An O-structure is an architectural memory element, providing versioning and renaming; synchronization is obtained using acquire-release semantics. Our extension slightly modifies the semantics from prior work, and adds fine-grain locking. The extended memory system not only provides direct synchronization of known dependencies, but also provide explicit locking, allowing MVM to handle missing dependencies.

Importantly, MVM targets imperative code. Traditional dataflow execution models targeted functional code; such code prohibits side effects, forcing the input and output of every task to be explicit as function parameters and return values. While making dependencies clear eases scheduling, the functional model relies on value semantics, causing data structures to be copied rather than shared. I-structure [8] and M-structure [9] were proposed to mitigate the cost of the copies, but neither resulted in the flexibility and efficiency of imperative code.

MVM is not bulletproof: mistreated side-effects can cause dependencies to be violated, resulting in undefined behavior. Unlike a model that relies on speculation and verification to ensure correct execution, programmers (and/or compilers) have to use MVM correctly. In this paper we'll also explore how this is done.

This paper focuses on the interaction between a task-based execution model and multi-versioned memory. Heavily researched aspects such as high-level programming model and optimized scheduling were minimally implemented and are left for future work. This paper makes the following contributions:

- Define MVM, a task-based programming and execution model using O-structures for fine-grained synchronization.
- Refine the interface and semantics of O-structures, a memory element allowing versioning, renaming and fine-grain locking.
- Demonstrate MVM's ability to efficiently parallelize operations on various pointer-based data structures, which can't be handled by existing task-based models.

II. Task-based programming and memory versioning

Non-speculative task-based programming models are effective in expressing parallelism, but are not backed by efficient hardware primitives that support the statically sequential property. These models must thus employ software-level primitives for synchronizing and communicating inter-task data dependencies, and thereby incur prohibitive runtime costs. In turn, programmers are forced to use coarse-grain data dependencies in order to amortize the runtime costs.

The emerging class of task-based dataflow programming models (e.g., [2], [10], [3], [1], [4]) demonstrates this tradeoff. These models extract parallelism by scheduling tasks using the dataflow firing rule. These models employ programmer annotations of task inputs and outputs in order to identify in-memory task dependencies. Task inputs and outputs are used to construct the task dataflow graph that guides the scheduler. Given task inputs and outputs, the scheduler can expose parallelism by scheduling tasks according to their explicit data dependencies. The dataflow model thus removes the programming burden of dealing with explicit synchronization primitives (e.g., mutexes, semaphores, transactional memory [11]).

A major limitation of programmer annotation of input and output parameters is that annotations are restricted to contiguous or strided [12] memory regions. This limitation is a culmination of two constraints. First and foremost, annotations need to be expressed at coding time, and must therefore be static. Furthermore, the expressed dependencies must be compact, since those that involve large memory segments or multiple tasks might curb parallelism. As a result, programmers cannot express dependencies that involve irregular, dynamic data structures (e.g., lists and trees), which greatly limits the problem domain that such programming models can address. This limitation gives programmers who wish to process irregular data structures no choice but to use fine-grain, explicit, synchronization primitives, such as mutexes or transactional memory. The nature of these primitives makes them more difficult to reason about.

A. The need for memory versioning

Task-based programming can provide a hierarchical code decomposition that is *statically sequential* [1] if it implicitly preserves the original, sequential ordering of operations. We argue that the models' inherent ordering can be leveraged to order memory operations.

Hierarchical task decomposition and memory order: Figure 1 demonstrates a hierarchical task decomposition and its mapping to a virtual timeline that represents the original program order. The figure depicts a decomposition of task T_0 (i.e., the original program) into three tasks, T_1, T_2 and T_3, which are further decomposed. In a sequential execution, each of the tasks views the memory in its state just prior to the task's execution. For example, task $T_{2,0}$ expects memory elements to be at virtual time vt_4. However, if a datum consumed by task $T_{2,0}$ is produced by task $T_{1,1}$, then a virtual time-cognizant memory system can identify that task $T_{2,0}$ consumes an item that is ready at virtual time vt_2 and can, therefore, execute it in parallel to T_1's remaining sub-tasks. Alternatively, if the compiler/programmer identifies that a datum produced by $T_{1,1}$ is not used by any of the remaining sub-tasks of T_1, it can notify the memory system to advance the virtual time of the said datum to task T_2 at vt_4 (horizontal arrow in Figure 1).

How versions remove these limitations: A memory system that allows data to be tagged with a version provides a fine-grained synchronization primitive, which needs not be explicitly specified as a task dependency. Since versions

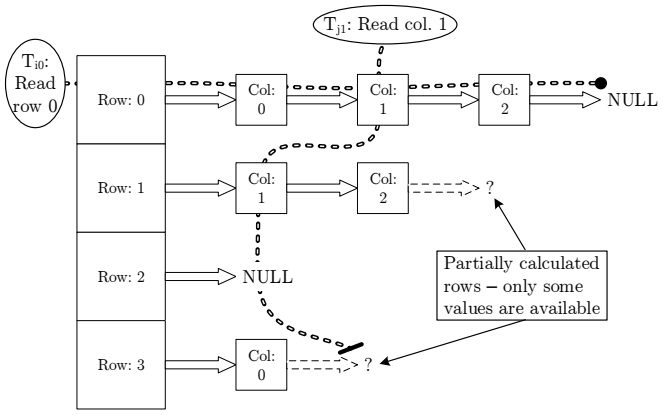

Fig. 2: A partially created 4*4 sparse matrix (in the List of Lists format). Task T_{i0} successfully reads row 0, while task T_{j1} stalls when trying to get the unavailable value of column 1 in row 4.

correlate to the sequential ordering of the program, the combination of data and version represents a producer-consumer relation. In a sequential run, the same memory location can serve multiple producer-consumer relations without causing data races. To achieve the equivalent ability in a parallel execution, each memory location can be renamed, namely store multiple versions of the same location. Importantly, only memory locations that form a dependency among tasks need be versioned and possibly renamed; task-local operations can use the conventional memory interface.

To illustrate the benefit of the proposed model, let us consider a series of operations involving sparse matrices. Figure 2 presents a partially created sparse matrix in the List-of-Lists (LIL) format. While it is being constructed, two tasks are trying to read the matrix: T_{i0} reads row #0, while T_{j1} reads column #1. Assuming each row was computed by a different task, the data's version corresponds to the row number. The two tasks operate as follows:

- Task T_{j0} uses version 0, and successfully reads the first row, which was fully calculated.
- Task T_{i1} uses a different version number for each row, and traverses the list until it finds the required column (or goes past it). Due to the fine-grained synchronization provided by versioned memory, the task successfully reads column #1 of row #1, even though the row itself has not been fully calculated yet. However, when reaching row #3, the task is stalled until the value is available.

Figure 2 demonstrates operations on a data structure that cannot be efficiently handled by traditional non-speculative task-based programming models. Since each row is represented by a dynamically created list in which nodes are anonymous, a dependency on a particular column could not be expressed at compile time. The complete row could be treated as a whole using coarse-grained synchronization, but then readers such as T_{j1} would have been stalled even when their data is already available. Speculative models can handle the unknown dependencies, but if the matrix is large, their read-set storage (whether implemented in L1 or dedicated structure) might overflow, causing a stall or squash.

It should be noted that the versioning in Figure 2 could be reduced to a single bit indicating whether the data is available or not. In section IV we describe more complex use cases, which use both versioning, renaming and per-version locking.

Summary: As previously noted, task-based execution models have been explored in depth (e.g., [13], [14], [2], [15], [3], [16]). All task-based execution models aim to execute tasks concurrently, and in some cases out of the original program order. The challenge, of course, is that an earlier (in program order) task produces values that a later (in program order) task needs. These producer-consumer dependencies fall into two broad categories: (1) those that can be statically known; and (2) those that cannot. Statically known dependencies, revealed either via the register dependencies in the control-dataflow graph, or through alias analysis of memory accesses, can be dealt with in one of two ways: (a) imposing a scheduling constraint that delays the later (consuming) task until the earlier (producing) task has created the necessary value; or (b) speculating that the producer-consumer relationship will not arise dynamically and/or speculating on the value the producer will write. **Our goal is to provide a third option that aids parallel execution of (1) known dependencies, by providing fine-grained inter-task synchronization via memory and (2) unknown dependencies, by providing concurrent storage of multiple values for the same memory location, along with semantics for how these values relate to each other and control inter-task synchronization.**

III. O-STRUCTURES

An O-structure is a memory element that maintains multiple versions of the datum. These versions are ordered. For example, given versions 1, 2 and 3, version 2 is ordered after version 1 and before version 3. We will see later that this order is expected to be used to reflect original program order for task-based execution models. Programs must specify what version of the memory location they wish to load or store. Initially, all versions exist in an *uncreated* state. 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 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 ever created. In such a situation, loads directed to version 1 will stall, while those directed to version 2 will succeed. We also introduce two new memory operations named LOAD-LOCK and UNLOCK, which enable ordering when dependencies cannot be statically analyzed in full. For brevity, we omit the address from the following list of O-structure operations. All operations, however, would also take an address just as with ordinary loads and stores. O-structures support the following operations:

- 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 by an older task $u < v$, the call stalls. If it is locked by a younger task $w > v$, then the call returns the value successfully.
- 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 by an older task, 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 a

version is created, its value must not be 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.

Task-based execution models should use O-structures selectively, as versioned operations do bear some space and time overhead comparing to conventional memory. Task-private memory accesses (e.g., to the stack) and fully analyzable data-parallel operations 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 program order and being non-speculative. Both conventional (unversioned) and O-structure (versioned) functionality should be architecturally visible. Conventional memory is accessed using traditional LOAD and STORE operations while versioned memory uses the dedicated operations described above.

The most straightforward way to use O-structures in a task-based execution model is to ensure that all tasks have a labeling that matches the original sequential program order. For example, suppose the programmer (or compiler) decomposed a loop such that each loop iteration was in a separate task. A straightforward task labeling would assign loop iteration n with task identifier n, iteration $n+1$ with identifier $n+1$, and so on. Nested spawning can also be supported by using multi-level identifiers (our approach), ranges [17] and other schemes [18]. A task could then use its task identifier as a version identifier when accessing an O-structure.

IV. Handling irregular data structures

One of the unique features of the MVM model is its ability to parallelize operations on a major class of pointer-based data structures. Such irregular data structures are notoriously hard to parallelize in conventional task-based models because (a) dependencies are unknown in advance and are anonymous, preventing them from being explicitly specified, and (b) breaking operations on large data structures into small speculative tasks is disadvantageous, yet large tasks might read more than a speculative runtime can handle. In this section we describe how O-structures can be used to allow large, non-speculative tasks to concurrently operate on a shared data structure.

A. Unipath, Semi-unipath and Multipath code

Data dependencies determine the amount of parallelism in an algorithm. Data parallel algorithms such as matrix multiplication, in which processing of data elements is mostly independent, are easy to accelerate on parallel execution environments. But in most ubiquitous data structures — lists, trees, graphs. and even sparse matrix representations — data elements are linked and are thus dependent. Those data structures have an irregular memory layout, which allows efficient lookup, dynamic resizing, and sometimes rebalancing. But once the layout becomes part of the data, the operation of reaching a certain element depends on previous operations that have constructed the linked path to the said element. Irregular data structures are thus harder to parallelize.

One of the main strengths of O-structures is their support for parallelizing irregular data structures. We found it useful to classify algorithms and the data structures they manipulate into three categories: unipath, semi-unipath, and multipath.

Unipath algorithms are series of operations in which a task accesses data in the following way: (a) entering the data structures is done via a single entry point; (b) each datum is reached by traversing a single (per datum) sequence of pointers; and (c) a task never revisits a datum once it has been passed through. Unipath algorithms are most common on irregular data structures with a single entry point such as trees or lists. Insertions, removals and lookups all start from the entry point and work their way to the required element by traversing the single path to it. Parallelizing unipath algorithms with O-structures can therefore rely on pipelining: a task can lock and unlock the memory locations that contain pointers, and so long as tasks enter the root of the data structure in program order, they will traverse the data structure in program order. This is demonstrated in the example below.

Multipath algorithms include a traversal that can either reach a node from two different pointer chains or revisit a node after the first access. An example of the former traversal is a breadth-first search executed on a graph. If the search is parallelized, traversals cannot be synchronized, and the order in which they will reach nodes is arbitrary. O-structures do not facilitate parallelizing multipath algorithms.

Semi-unipath algorithms can be partially parallelized. Consider insertion into a balanced red-black tree: a mutator must wait for the previous mutator to complete the rebalance phase, because the rebalance might modify the head. If it enters earlier, the concurrent modifications of the tree might yield an erroneous state. Read-only tasks, on the other hand, can run in parallel with the previous mutator, following it on its way to the mutation point. If they look up the newly inserted node, they will find it in a possibly unbalanced tree. The imbalance, however, will not be by more than one node. Parallelization of semi-unipath algorithms resembles the use of a reader-write lock, as it allows multiple readers to run in parallel while writers are serialized. However, versioning allows the writer to run before previous readers have completed, and readers to run in parallel with the writer.

It should be noted that the above classification relies mostly on the algorithm. While the data structure must provide a single entry point to allow unipath operations, the traversal dynamics determine if the unipath property holds. For instance, a singly linked list seems natural for unipath operations, but if the algorithm holds pointers to different nodes at the same time, it might revisit a node it had already passed. On the other hand, a forward-only traversal on a doubly linked list will be considered a unipath operation even though the data structure allows easy reversing.

B. Example

Listing 1 illustrates how O-structures can be used to allow parallel execution of pointer-heavy tasks. Using manually inserted task spawning and O-structure access primitives[1], the

[1]Compiler support can relieve the programmer from using the low-level interface directly and require only hints to incorporate versioning.

Listing 1: List insertion using hand-over-hand locking

```
1  insert_end(root, n, curTask) {
2    prev = root; // assume non-null
3    // reserve a specific version of the head
4    // (stalls if does not exist)
5    cur = LOCK-LOAD-VERSION(root->next, curTask);
6    while (cur != nullptr) {
7      // reserve latest version of next pointer
8      next = LOCK-LOAD-LATEST(cur.next, curTask);
9      // make prev. ptr available to next task
10     UNLOCK-VERSION(prev.next, curTask);
11     prev = cur;
12     cur = next;
13   }
14   // create a new version for the new node
15   STORE-VERSION(prev.next, n, curTask);
16 }
17 // the spawning loop
18 while (more_work_available())
19   spawn_task(insert_end, root, get_work_item(),
         taskid++);
```

code implements insertions to the end of a singly linked list; those insertions are *unipath* operations.

1) Tasks enter the insertion routine and LOCK-LOAD-VERSIONs the root node of the data structure (line 5; the specific version of the root is known because of the ordered way in which tasks enter the routine). This provides an initial ordering, which is maintained by the following steps.
2) Traversal over internal pointers is done using hand-over-hand locking [19] (lines 8, 10). Due to the ordered entry and the unipath properties, read-after-write dependencies are thus enforced in internal nodes as well. Importantly, the latest versions of each pointer are accessed, since the specific versions are unknown.
3) The use of STORE-VERSION (line 15) eliminates write-after-read dependencies, namely inserts need not wait for preceding read-only tasks. Readers can thus avoid using locks during traversal. Omitting versioning and relying only on locking would require readers to lock as well, adding overhead and reducing parallelism.
4) While many versioned pointers might be locked during an insert, only a single version is created. Locking does not produce new data, hence need not create new versions.

V. Evaluation

In this section, we evaluate the MVM model's ability to parallelize operations on common pointer-based data structures: linked list, binary tree, hash table and red-black tree, and on two non-unipath algorithms: matrix multiplication and Levenshtein distance. We have not yet completed our compiler implementation effort, so our evaluation is confined to these microbenchmarks. Nevertheless they reveal both the potential and limitations of O-structures.

A. Methodology

We implemented an O-structure memory system in the gem5 [20] simulator executing in system emulation (SE) mode. The implementation michroarchitecture is beyond the scope of this paper, but was modeled in detail. The system configuration is presented in Table I. The task scheduler was implemented in software and used a static assignment of tasks to cores. This policy imposes a minimal runtime overhead, but does not support load balancing.

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 I: The experimental platform

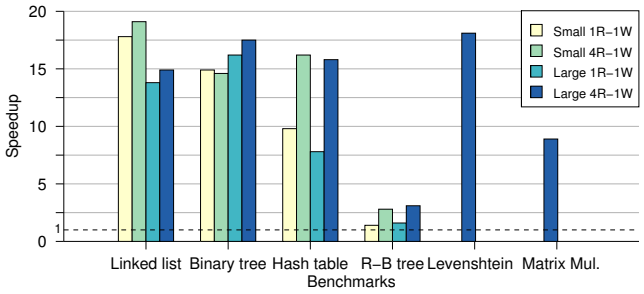

Fig. 4: Speedup of parallel versioned (32 cores) over sequential unversioned. Small benchmarks start with 1000 elements and large with 10000; Read-intensive (4R-1W) is 4 reads per write and Write-intensive (1R-1W) is 1 read per write.

Our evaluation of the unipath and semi-unipath data structures (sorted linked list, binary tree, hash table, and red-black tree) interleaves lookup, insert and delete operations in different ratios (the data structures were pre-populated). 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 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. Figure 3 and Figure 5 present the scalability (self-speedup) of the measured benchmarks given a varying number of cores. However, versioning adds overhead; the real benefit is obtained by comparing parallel versioned to sequential unversioned code. Figure 4 thus presents the maximal speedup of *versioned vs. unversioned* code running all sizes and ratios.

B. Speedup analysis

Linked list benchmarks demonstrate near-linear self-speedup (Figure 3). A major reason is the inherent poor locality of the algorithm and a memory footprint that exceeds the L1 cache. Since the Last-level cache (LLC) must be accessed often (over 50% L1 miss rate on small 1- and 32-core runs), coherence overhead is *relatively* smaller with O-structures. In addition, the list is long enough to accommodate a large number of tasks operating concurrently. Interestingly, on 32-core runs, slow-moving mutating tasks rarely stall

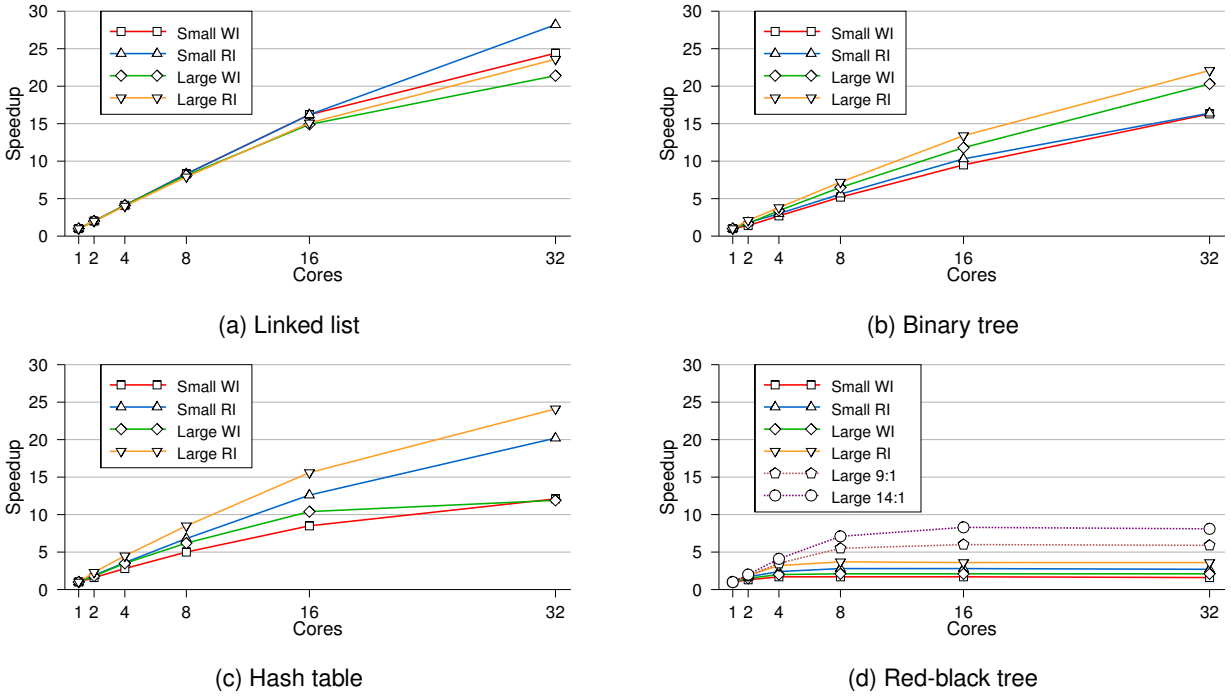

Fig. 3: Scalability of unipath algorithms (comparing to a single-core *versioned* run). Workloads differ by size and operations mix (Read-Intensive is 3 reads per 1 write; Write-Intensive is 1 read per 1 write)

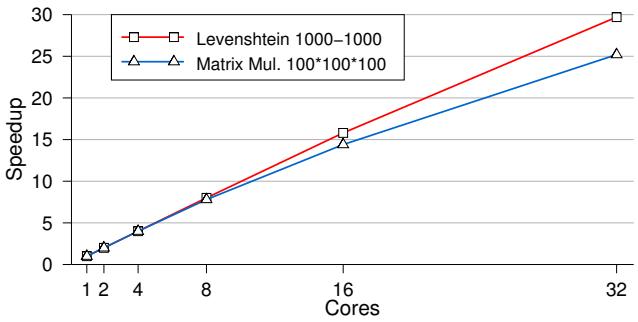

Fig. 5: Scalability of regular algorithms (comparing to a single-core *versioned* run).

(∼15% of loads) due to ordering, while fast-moving read-only tasks are frequently stalled trying to bypass older mutators. Finally, even though unversioned runs do not have versioning overhead, for all settings the unversioned execution is under 1.6× faster on a single core. This effect is mostly due to high L1 miss rate. In summary, the key takeaways from this analysis are that (1) the inherently poor locality masks the overhead of cross-core communication; and (2) for sufficiently large lists, true memory dependencies do not limit parallelism. O-structures enable almost arbitrarily high speedup for task-based execution.

Binary trees have a larger memory footprint than a linked list but retain much better temporal locality. Scalability of small runs suffers from a reduction in L1 hit rate (94% on 1 core; 88% on 32 cores), also evident in the increase of versioned load latency (∼4×; such loads comprise ∼20% of versioned memory accesses). Large runs show better scalability due to a much lower rate of ordering synchronization, which occurs mostly at the top of the tree. For example, the average rate of loads that stall due to ordering on a large, 32-core run is ∼25%, while ∼40% stall on an small run. As opposed to ordering stalls, which imply a partial serialization of the execution, conventional capacity cache misses are handled in parallel, and do not effect scalability. In summary, the key takeaways are that (1) given reasonable cache sizes, versioning overhead does not impact performance due to the increased locality in a tree compared to a list; (2) larger trees result in less synchronization stalls and (3) for really large trees, the increasing rate of capacity misses reduces performance.

Hash tables store buckets in an array, which has a regular memory layout. Each bucket holds a linked list, which is an irregular data structure, shown to be effectively parallelizable using O-structures (above). To obtain the unipath property, the pointer to the table is used as the ordering point, and each list is then synchronized independently. Properly used hash tables have a load factor (avg. nodes per bucket) of about 1, which leaves little room for pipeline-style parallelization. To keep our benchmarks realistic, a load factor of 1 was used.

Hash table benchmarks with a high ratio of mutating tasks do not scale well (Figure 4). This is expected, as every other task locks the head (i.e., table pointer),forming a bottleneck. The versioned load miss rate rises up to 85%. Yet even operations on empty buckets are non-trivial: inserting a node requires locking the bucket, allocating a node, setting the node's value and next fields, and creating a new bucket version to point to that node. This produces 6 memory writes; adding unversioned operations such as hash calculation and memory allocation yields a sufficiently deep pipeline. That pipeline provides reasonable speedup. The key takeaway is that operations on data structures that intuitively seem inappropriate for pipelining may be long enough to scale well after all.

Red-black trees are semi-unipath data structures. Mutating tasks are serialized, and read-only tasks can follow the steps of the previous mutator, but only view the unbalanced state. Parallelism is therefore strictly bound to a single writer plus the readers that come between it and the next writer: 2 on a 1:1 ratio, 5 on a 4:1 ratio and so on. Moreover, our remove algorithm sometimes removes the node after effectively rebalancing the tree, allowing almost no readers-writer parallelism. Running only inserts and lookups increases the speedup by 3-10%, depending on the initial rate of remove operations. Read-oriented workloads yield a much better speedup – 6× for a 9:1 ratio and 8× for a 14:1 ratio. The key takeaways from our study of red-black trees is that (1) O-structures can be used to parallelize and speed up balanced tree operations, but scaling is limited; and (2) some operations that can be parallelized (such as remove) require manual modifications that would be challenging for a compiler to infer.

Matrix multiplication and Levenshtein distance are mostly data parallel algorithms and scale well. While the unipath pattern could not be applied, hand-parallelization was simple. Neither algorithm requires renaming, and both use O-structures as I-structures [8]. It should be noted that a single core matrix multiplication runs 2.5× slower than the unversioned implementation due to the large number of O-structures and versioned operations. The key takeaway is that although their full set of features might add noticeable overhead and is not fully utilized, O-structures can be effectively used to parallelize ubiquitous non-unipath algorithms.

Irregular data structures generally make suboptimal use of some central micro-architectural mechanisms: large memory footprints overflow private caches; irregular layouts forbid prefetching; and data-dependent conditional branches yield a high rate of mispredictions [21]. O-structures do not solve any of these problems, but the inherently low IPC rate significantly reduces the relative cost of versioning and parallelization.

VI. Related Work

A. Non-speculative task-based programming models

A plethora of non-speculative task-based programming models and runtimes have been proposed. Most of those models belong to one of three broad classes: (1) those with no support for inter-task dependencies; (2) those with rigid implicit dependence relationships; and (3) those that support programmer annotations that explicitly express inter-task dependence relationships. OpenMP 3.0 [22], [15] and Grappa [23] are examples of runtimes in the first category, as they do not support inter-task dependencies in the runtime. Programmers must explicitly use synchronization primitives (e.g. mutex, barrier) and be mindful of the limitations and scheduling behavior of the runtime. Cilk [13], [14] is an example of a runtime in the second category; child tasks are forced to complete before the parent task. Finally, CellSs [2], StarSs [24], TaskSs [3], OmpSs [25] and OpenMP 4.0 [4] are examples of runtimes in the third class; programmers annotate task inputs and outputs, and tasks are restricted to executing only when their inputs are ready. Less conventional models such as ADF [26] combine partial dependence specification with transactional memory, handling unspecified shared state.

While most task-based programming models rely on simple data types (scalars and arrays) for input and output specifications, some opt for a richer interface based on streams. Gordon et al [27] combined the StreamIt language [28] with a task-based model in which streams provide communication channels between tasks; the streams support push, pop and peek operations. OpenStream [5] extended streams with sliding windows, broadcasts and more. Streams simplify communication of large amounts of data among cores and allow unnamed data to be shared. Still, streams themselves must be explicitly specified, and their expected use known at compile time.

B. Thread-level speculation (TLS)

Thread-level speculation is a technique in which multiple threads operate in parallel, and the mutations of each thread are committed to the shared state only if they are consistent with respect to the previous state. While the use of coarse-grained speculation resembles that of Transactional Memory [11], 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 the order of commits.

TLS has yielded many implementations (e.g., [29], [30], [31], [32], [33], [34], [35], [36], [37], [16]). MultiScalar [29], 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 [38] for aliases, and an innovative structure called Speculative Versioning Cache (SVC) [39] in order to scale past the limits of centralized store-queue designs.

TLS and MVM both target sequential programs and use versioning as part of the implementation, but the programming models and execution strategies 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 to express her knowledge of the program and the available parallelism. By leveraging programmer knowledge, task-based programming models using MVM 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 MVM relies on known dependencies or patterns such as unipath algorithms. However, TLS implementations are limited in the number of the data dependencies they can track, whether using dedicated structures (e.g., [29], [31], [16]) or L1 (e.g., [39], [32], [33], [34], [37], [36]) to store speculative state and track access to it. O-structures, on the other hand, can track versioning at memory-scale, eliminating task size limit.

C. Fine-grained pipelining

Pipelining has been suggested as a way to handle irregular algorithms and data structures by systems such as DOACROSS [40], DSWP [41], HELIX [42] and recently BDX [43]. Those works excel at parallelizing tight loops, intensively operating on the data structure (namely, each and every entry is not only accessed but processed). MVM allows a wide range of irregular data structures to be shared; pipelining merely ensures read after write dependencies. Moreover, tasks that become non-dependent (e.g., traversing different tree branches) are not ordered once they diverge.

D. Memory versioning

The concept of implicitly storing multiple versions of a memory element for synchronization and checkpointing is common in *transactional memory* [11] (TM), *thread-level speculation* [29], [44], [32], [36], [16] (TLS) and some task-based systems (e.g., Swam [45]). While they demonstrate the feasibility of extending memory with versions, none expose versioning to the program, and none provide the set of features used by MVM.

VII. Conclusions

This paper introduces the MVM task-based execution model. Building on prior work on memory versioning and renaming [7] this runtime efficiently parallelizes pointer-heavy data-structures *non-speculatively*. By slightly changing the previosuly proposed O-structure semantics to add fine-grained version locking, we were able to achieve up to 19× performance improvement on a simulated 32-core system. This speedup is encouraging and suggests that, while writing parallel software remains a challenge for pointer-rich code, architecture support for novel memory concepts such as versioning and renaming can unlock performance hidden inside such software.

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] G. Gupta and G. S. Sohi, "Dataflow execution of sequential imperative programs on multicore architectures," in *MICRO*, 2011.
[2] P. Bellens, J. Perez, R. Badia, and J. Labarta, "CellSs: a programming model for the Cell BE architecture," *Supercomputing*, Nov 2006.
[3] 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 *MICRO*, Dec 2010.
[4] OpenMP Architecture Review Board, *OpenMP Application Program Interface Version 4.0*, Jul 2013.
[5] A. Pop and A. Cohen, "Openstream: Expressiveness and data-flow compilation of openmp streaming programs," *TACO*, vol. 9, Jan. 2013.
[6] R. M. Karp and R. E. Miller, "Properties of a model for parallel computations: determinacy, termination, queueing," *SIAM J. Applied Mathematics*, vol. 14, Nov 1966.
[7] E. Gilad, E. W. Mackay, M. Oskin, and Y. Etsion, "O-structures: Semantics for versioned memory," in *MSPC*, 2014.
[8] Arvind, R. S. Nikhil, and K. K. Pingali, "I-structures: data structures for parallel computing," *TOPLAS*, vol. 11, Oct. 1989.
[9] P. S. Barth, R. S. Nikhil, and Arvind, "M-Structures: Extending a parallel, non-strict, functional language with state," in *Functional Programming Languages and Computer Architecture*, 1991.
[10] J. Perez, R. Badia, and J. Labarta, "A dependency-aware task-based programming environment for multi-core architectures," in *CLUSTER*, Sep 2008.
[11] M. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," in *ISCA*, 1993.
[12] J. M. Perez, R. M. Badia, and J. Labarta, "Handling task dependencies under strided and aliased references," in *ICS*, Jun 2010.
[13] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *PPoPP*, 1995.
[14] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *PLDI*, 1998.
[15] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, "The design of OpenMP tasks," *TPDS*, vol. 20, no. 3, 2009.
[16] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez, "A scalable architecture for ordered parallelism," in *MICRO*, 2015.
[17] C. Segulja and T. Abdelrahman, "Architectural support for synchronization-free deterministic parallel programming," in *HPCA*, 2012.
[18] J. G. Cleary, J. A. D. McWha, and M. Pearson, "Timestamp representations for virtual sequences," in *PADS*, 1997.
[19] R. Bayer and M. Schkolnick, "Concurrency of operations on B-trees," *Acta Informatica*, vol. 9, Mar 1977.
[20] 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*, vol. 39, no. 2, 2011.
[21] O. Green, M. Dukhan, and R. Vuduc, "Branch-avoiding graph algorithms," in *SPAA*, 2015.
[22] OpenMP Architecture Review Board, *OpenMP Application Program Interface Version 3.0*, May 2008.
[23] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin, "Latency-tolerant software distributed shared memory," in *Usenix ATC*, 2015.
[24] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta, "Hierarchical task-based programming with starss," *IJHPCA*, vol. 23, Aug. 2009.
[25] A. Fernández, V. Beltran, X. Martorell, R. M. Badia, E. Ayguadé, and J. Labarta, "Task-based programming with ompss and its application," in *EuroPar*, 2014.
[26] V. Gajinov, S. Stipic, O. S. Unsal, T. Harris, E. Ayguadé, and A. Cristal, "Integrating dataflow abstractions into the shared memory model," in *SBAC-PAD*, 2012.
[27] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *ASPLOS*, 2006.
[28] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "Streamit: A language for streaming applications," in *CC*, 2002.
[29] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar processors," in *ISCA*, 1995.
[30] 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," tech. rep., Stanford Univ., 1997.
[31] H. Akkary and M. A. Driscoll, "A dynamic multithreading processor," in *MICRO*, 1998.
[32] V. Krishnan and J. Torrellas, "A chip-multiprocessor architecture with speculative multithreading," *IEEE TC*, vol. 48, Sep 1999.
[33] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry, "A scalable approach to thread-level speculation," in *ISCA*, 2000.
[34] M. Cintra, J. F. Martinez, and J. Torrellas, "Architectural support for scalable speculative parallelization in shared-memory multiprocessors," in *ISCA*, 2000.
[35] 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 *ICS*, Jun 2005.
[36] 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," *TPDS*, vol. 19, Jul 2008.
[37] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry, "The STAMPede approach to thread-level speculation," *TOCS*, vol. 23, Aug. 2005.
[38] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi, "Dynamic speculation and synchronization of data dependences," in *ISCA*, 1997.
[39] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi, "Speculative versioning cache," in *HPCA*, 1998.
[40] R. Cytron, "Doacross: Beyond vectorization for multiprocessors," in *ICPP*, 1986.
[41] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August, "Decoupled software pipelining with the synchronization array," in *PACT*, 2004.
[42] S. Campanoni, T. Jones, G. Holloway, V. J. Reddi, G.-Y. Wei, and D. Brooks, "HELIX: Automatic parallelization of irregular programs for chip multiprocessing," in *CGO*, 2012.
[43] D. C. S. Lucas and G. Araujo, "The batched doacross loop parallelization algorithm," in *HPCS*, 2015.
[44] L. Hammond, M. Willey, and K. Olukotun, "Data speculation support for a chip multiprocessor," in *ASPLOS*, 1998.
[45] H. Vandierendonck, G. Tzenakis, and D. S. Nikolopoulos, "A unified scheduler for recursive and task dataflow parallelism," in *PACT*, 2011.