O-structures: Semantics for Versioned Memory

Eran Gilad
Technion
erangi@cs.technion.ac.il

Eric W Mackay Mark Oskin
University of Washington
emackay@cs.washington.edu
oskin@cs.washington.edu

Yoav Etsion
Technion
yetsion@tce.technion.ac.il

Abstract

This paper introduces O-structures, a novel architectural memory element that can be used to facilitate parallelism in task-based execution models. Much like register renaming, each write to an O-structure creates a new version of program memory at that location. These versions can be accessed concurrently and out of program order. O-structures provide a set of semantics that match the needs of task-based execution models, specifically allowing tasks to synchronize on specific versions of memory as well as coordinate access when the necessary version is not known at compile time.

In this work, we describe O-structures and provide their complete semantics. We also discuss how a task-based execution of basic data structure manipulations on common data structures (arrays, lists, trees, etc) operate. Results are presented that measure the exposed memory-level parallelism (MLP) in these operations. We find that for previously difficult to parallelize data-structures, such as linked lists, binary trees and sparse-matrix codes we see significant memory level parallelism (50-100 operations per cycle) when using O-structures.

Categories and Subject Descriptors B.3.2 [*Memory Structures*]: Design Styles

Keywords O-Structures, Memory renaming, Out-of-Order

1. Introduction

This paper introduces *O-structures*, a new type of storage element designed to aid the parallel execution of sequential programs. We choose the name "O-structure" because it provides *ordered* access to multiple versions of program state for the same memory address. O-structures are inspired by I-structures [3] and M-structures [5] for dataflow machines, but carry semantics designed for task-based execution models of imperative, not dataflow programs. Like I-structures, O-structures only allow versions to be written once, like M-structures, they provide lock-like semantics to access memory versions across tasks. Novel to O-structures, however, is their ability to support ordered access to multiple values stored within them.

O-structures are designed to be coupled with task-based execution models, where a single imperative program thread is carved via software or hardware into multiple program tasks. Tasks are expected to be *ordered*, yet crucially it is not expected that tasks are spawned in sequential order [26]. Like the von Neumann model, in task-based execution models, state is manipulated sequentially within a task and persists indefinitely. Like the dataflow model, however, tasks that write state essentially create new versions of that state, rather than overwrite the existing data in memory which may be needed for the completion of tasks earlier in program order; and also like the dataflow model state can be created in parallel across program tasks. Finally, like the multithreaded von Neumann model, there exists a precise set of semantics for how memory operations interact between tasks.

O-structures are inspired by register renaming. Each time a task writes an address a new *version* of that memory location is created. Throughout this text, we denote the pairing of an address and a version of data at that address by <address:version>. The intent is, just as with register renaming, accesses across tasks are not serialized because of false output-address dependencies. When multiple stores are in flight to the same memory address, those stores will be directed to different versions and can thus proceed in parallel. When multiple reads are in flight to the same address, those reads will be satisfied by the appropriate version written by a previous store, or held until the matching store is complete.

By resolving the ambiguity in naming different generations of a single memory datum, memory versioning increases task parallelism by allowing tasks to both spawn out of program order and generate state concurrently. Without memory versioning, correct association of data producers and consumers requires either conservative code generation, which inhibits memory-level parallelism [31], speculation [22], or restricted, in-order spawning of tasks [6, 12].

The focus of this paper is the semantics of O-structures, their interface and their interaction with task execution environments. While not describing a concrete implementation, we argue for the use of O-structure semantics as a building

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

MSPC '14, June 09–11, 2014, Edinburgh, United Kingdom.
Copyright © 2014 ACM 978-1-4503-2917-0/14/06. . . $15.00.
http://dx.doi.org/10.1145/2618128.2618130

block and discuss how O-structures can be used to construct compound data structures. We demonstrate the potential parallelism extracted by O-structures using an execution-model simulator. This paper makes the following contributions:

- **Introduce:** This is the first paper to describe O-structures, a novel architectural memory element designed for task-based execution models.
- **Define:** We describe the semantics of O-structures and motivate them by way of example.
- **Use:** We describe how common data structures (arrays, lists, trees, etc) can use O-structures to gain parallelism in a task-based execution model.
- **Evaluate:** This paper contains the results of a high-level execution model simulation that demonstrates the memory-level parallelism (MLP) that O-structures unlock.

We begin in the next section by motivating the need for a new memory element. Next in Section 3 we formally define the behavior of O-structures. In Section 5 we demonstrate how O-structures can be used to build common data structures. Section 6 evaluates the MLP unlocked by O-structures. Finally, we conclude in Section 7.

2. Motivation and background

2.1 Existing memory interfaces

von Neumann systems: For single threaded von Neumann-based execution model machines, memory semantics are quite straightforward. Read operations must supply an address and the result is whatever the value of the previously written Write operation to that same address was. At the hardware level (below the operating system and user process abstraction) a Read sent to an address that has no prior Write returns an undefined value. There is no accepted name for a memory location on a von Neumann machine, so as a shorthand in this paper we will call them V-structures.

When multiple processors (or cores) are sharing access to the same physical memory, additional semantics are enforced. This is known as the consistency model and it defines how Read and Write operations are interleaved across cores. Additional operations, fences, are typically provided in order to allow programmer control of the interleaving. In addition, atomic operations, such as Atomic-Compare-and-Swap, or Test-and-Set, are introduced for applications to synchronize their threads of control with changes in memory state.

Dataflow systems: With dataflow computers [11] the primary memory system is the *token store*. Conceptually, the token store is an associative array. Programs name data values with a *tag*. The <tag:value> is inserted into the token store by the result of an instruction. Dataflow machines have been designed and built with a wide variety of tag and matching schemes, but from a high level perspective tags are used to name the inputs to a particular dynamic instance of an instruction to be executed. Once all inputs are available (a tag match), the instruction is executed.

Early on researchers recognized the inefficiency of building large token store memories [9] and developed alternative "dataflow-compatible" memory systems. Perhaps the earliest and most widely known is the I-structure [3]. An I-structure is a memory location that can be written only once but read multiple times. When a Read operation is dispatched to an address, if no Write to that address has occurred, the Read request is queued. When a Write operation occurs to an address, any pending Read requests are dequeued and provided with the written value. Subsequent Read operations to the address are satisfied with the result of the Write. A subsequent Write to the same address is an error. The I-structure was followed up by the M-structure [5]. An M-structure is subtly different than an I-structure, and conceptually an M-structure is a set data-structure. A Write (called a *put* in M-structure parlance) inserts a value to this set. A Read (*get*) removes a value. When the set is empty, a pending Read is queued until it can be satisfied by a Write.

Exotic systems: In addition to V-structures, token stores, I-structures and M-structures, systems have been built with other novel memory semantics. The Cray MTA [2] had full/empty bits on each memory location. These bits eased inter-thread synchronization by providing more sophisticated read and write operations, such as Read-when-Full, and Write-when-Empty. Several systems have also incorporated basic type information into the memory system [10, 23]. Machines have been built that support in-memory operations such as Fetch-and-Add. When these operations are associative they can even be combined [15]. Finally, several researchers have proposed more sophisticated in-memory computation logic, most notable in the late 1990's with the intense amount of research into "Intelligent Memory" architectures [8, 17–19, 24].

2.2 Limitations of existing memory interfaces

Our goal is to have a memory system that supports highly concurrent execution of single-threaded programs written in imperative languages. These execution systems, typically task-based [6, 13, 20, 27, 30], have two defining characteristics to note here: (1) their goal is to execute with high concurrency, hence ideally multiple memory operations to the same address will be in flight simultaneously; and (2) because instructions (or tasks) come from an imperative language program thread, there is an ordering between them. This means the outstanding memory requests issued by those tasks have an ordering as well, and this ordering must be respected for correct program execution.

A V-structure (von Neumann memory system) is not a good match for these execution models, because only one value for a given address can be stored at a time. Moreover, there is no ordering between Read and Write operations. Without additional logic, a Write later in program execution could erroneously satisfy an earlier Read. The MultiScalar team recognized this issue and developed the Speculative Versioning Cache [14]. Speculative Multithreading systems also rely on processor caches [20] to maintain different versions of data for the same address. Recently, non-speculative versioning systems, which rely on programmer annotations of task read and write sets, have been proposed. Task superscalar [12] uses memory renaming to eliminate output dependencies in task-based execution. ROKO [29] maintains serial semantics using read and write timestamps for every

shared address, but only a single version of the data is kept. While simplifying versioning implementation, this does not eliminate output and anti-dependencies, and requires that read operations update a versioning data structure.

I-structures are also a poor fit for task-based execution models because they can only be written once. I-structures are really not designed for programs written in imperative languages. While it is theoretically possible to compile such programs to a system that uses I-structures for memory storage, the effect will be to grossly waste resources.

M-structures have some of the necessary aspects of a memory structure suitable for task-based execution models. M-structures can be written to more than once and can pair individual Write operations to Read operations. Read operations to an empty M-structure also block, which is one of the necessary semantics needed in order to stall Read requests from later tasks from completing until earlier Write's are dispatched. However, M-structures lack several key semantic elements. Write and Read operations are not ordered. For task-based execution models, this means a Read operation may end up being satisfied by a Write later in program order – again, clearly an error. Moreover, it's important in these models that potentially multiple Read operations be satisfied by the *same* Write operation.

2.3 O-structures

In the next section we provide a precise description of the semantics of an O-structure. Here, we will describe its operation informally, and motivate its use by way of example.

An O-structure supports multiple *versions* of the data stored within it. These versions are *ordered*. For example, version #1 is before version #5. Versions can be written out of order because they are independent of each other.

Programs must acquire permission to read and write a specific version of data, and they must release access to versions they have previously acquired write permission for. When a program requests permission to write a specific version of data, the O-structure immediately grants that permission and internally creates a new location to store the data for that version. A read permission is granted only if the required version has been released, otherwise the read blocks.

Programs release their permission to write a given version to a range of versions inclusive from the version being written to a specified version. For example, if a program has write access to version #5 and then releases this permission to version #9, then it has also released it to versions #6, #7 and #8. The release can be bound or unbound. A bound release is done to a specific version. An unbound release is done to the special version we denote θ. A θ-Release allows any future reader to acquire the released version. A simple use for θ-Release are immutable variables, but it is also instrumental when dealing with dynamic data structures, as presented in Section 5.

Let's consider now an extremely simple example (Figure 1). In this example, there are four tasks, some of which may execute concurrently. Given the semantics of O-structures these tasks compute the correct value for x, y and z as if the program code was executed sequentially. No other synchronization is required between these tasks. Task 1 writes a new value to variable x at version #1. Task 3 also creates a new value for variable x but with a different version #3. Task 2 reads the variable x and produces the variable y in version #2. Because Task 1 only releases the value of x to version #2 and not to version #4 if task 4 were to execute before Task 3 it would block on the attempt to acquire read access to version #4 of the variable x. Similarly, if task 3 were to finish before task 1 was even started, task 2 would still block on the attempt to acquire read access to version #2 of the variable x because task 3 only releases version #3 to be read by versions #4 and #5.

3. Semantics

3.1 Operations on O-structures

The following operations are used by application programs to manipulate memory state:

- **Write(address, version, data)** - write the given memory location and version with the given data.
- **Read(address, version)** - read the given memory location and version.
- **Acquire(address, version, read — write)** - acquire the permission to read or write the specific version of a given memory location. Programs specify only one type of access right at time, although multiple invocations of Acquire are permitted; i.e., read then write, or write then read.
- **Release(address, version_from, version_to)** - release the acquired permission for write access of version_from to version_to. The specified version_to can be a specific version, indicating a range of versions from version_from to version_to (inclusive), or a special version θ which indirectly specifies the next version written.

The semantics of Read and Write operations follow their customary definitions, except that a specific version is read or written. These operations may fault if the program has not acquired sufficient privileges to perform the operation on the specific version, however.

3.2 Semantics of Acquire

The semantics of acquire and release require further discussion. When an Acquire operaton is sent to the memory system for access to a specific version of an address, a requested privilege must also be specified. These privileges have the following meanings:

- **read** - When the read privilege is specified the value that is bound to the specified version is that of the *most recent write in version order*. This operation can block if that value has not yet been released to this version. Note that the most recent write may not be known at the time of issuing the Acquire operation; hence, this operation can block until it is known and released (see Section 3.3).
- **write** - Programs specify they wish to write a specific version of program memory. Once write privileges have been granted to a version of a memory location, subsequent reads of that location that depend on that version will block until the privilege has been released to them. The privilege to write a given version may only be granted once.

```
x = 5

y = x + 1

x = 6

z = x + 2
```

(a) Code

```
task_1: Acquire(x, write, #1)
        Write(x, 5, #1)
        Release(x, #1, #2)
task_2: Acquire(x, read, #2)
        Acquire(y, write, #2)
        Write(y, Read(x, #2) + 1, #2)
        Release(y, #2, #5)
task_3: Acquire(x, write, #3)
        Write(x, 6, #3)
        Release(x, #3, #5)
task_4: Acquire(x, read, #4)
        Acquire(z, write, #4)
        Write(z, Read(x, #4) + 2, #4)
        Release(z, #4, #5)
```

(b) "Compiled" code to use O-structures

Figure 1: Simple example code and how a task-based execution model can use O-structures to parallelize it.

3.3 Semantics of Release

When programs release write privileges to a given version and memory location they also specify a version (version_to) to release to. There are two possibilities for version_to:

- **A specific range:** If version_to is a specific version then all versions from version_from to version_to (inclusive) will be released to. This means that any attempts to acquire read access to those versions will proceed.
- **θ:** The special version θ is used to indicate "up to the next write". Sometimes the next write is *not* statically known. When an Acquire for read privilege is dispatched to the memory system for a particular version, and the most recent Release operation has released the version to θ then that Acquire operation proceeds immediately. The value of that requested read becomes the value of the write that was released to θ. It is a compiler (or program) error for a subsequent Acquire for write permissions to arrive with a version between the write that has been released to θ and an Acquire for read that has been satisfied by this write. Note that Acquire for read access to a version that is satisfied by a write that has been released to θ leaves the release to θ in place. That is, another subsequent Acquire for read operation for a version *greater* than the previous Acquire for read operation is immediately satisfied by the write that has been released to θ.

4. Task-Based Execution and Versioned Memory

Task-based execution parallelizes sequential programs by hierarchically decomposing them into short, sequential tasks that can execute in parallel. In this section we discuss the ordering semantics of tasks and how O-structures can provide synchronized and deterministic task memory accesses while enabling task schedulers to both spawn and execute tasks out-of-order.

4.1 Background: task-based execution models

Task-based models rely on programmer annotations of procedure calls that can execute asynchronously from the calling context[1]. This decomposition breaks a sequential program into a task tree. The backend runtime system then resolves the inter-task dependencies and extracts task parallelism whenever possible. The promise of task decomposition has prompted numerous task-based programming runtime environments [1, 4, 6, 7, 13, 21, 25, 27, 28].

Figure 2 presents a task decomposition. The code in Figure 2a, which updates and accesses a simplified key-value store, executes procedure calls as asynchronous tasks. The resulting task hierarchy is presented in Figure 2b.

Parallel execution of tasks requires that inter-task dependencies be conveyed to the task scheduler, so data dependencies will not be violated. Different tasking models employ different synchronization methods. These can largely be categorized as *execution-based synchronization*, which rely on execution barriers [4, 7, 13, 25, 28], and *data-based synchronization*, in which programmer annotations of task inputs and outputs are used to construct a data dependency graph [1, 6, 21, 27].

A key property of task-based decomposition is that tasks preserve the ordering of the original sequential code: An in-order traversal of the task tree recreates the program's original, sequential ordering of events, and in particular the original order of memory accesses. Tasks are therefore considered to be *statically sequential* [16].

Spawning a task splits the parent task into a prologue and epilogue parts. Everything that occurs in the parent before the task spawn should be ordered before the spawned task. The parent code that precedes the spawn becomes the prologue task, and should be ordered after the spawned task.

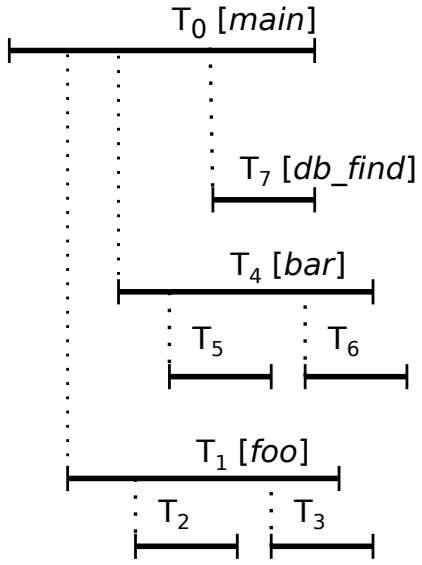
4.2 How task-based models can use O-structures

In essence, O-structures provide communication channels between tasks, forming an implicit data-based synchronization mechanism. Each version is a channel that has a single producer, which is the task that created it, and a well-defined range of possible consumers, which are the tasks that read the version. Using task identifiers as O-structure versions is the natural way to define such a communication channel (static versioning, which is used in Figure 2 for simplicity,

[1] For brevity, we assume that tasks are composed of complete procedures. Nevertheless, some models allow arbitrary code sequences to be executed as asynchronous tasks [4].

```
int db[N];
void db_update(int key, int val) { // version from, to
  Acquire(&db[key], Write, from);
  Write(&db[key], to, val);
  Release(&db[key], from, to);
}

int  db_find(int key) { // version from
  Acquire(&db[key], Read, from);
  return Read(&db[key], from);
}

void foo() { // version from, to
  spawn db_update(3, 20); // from=0, to=2;
  spawn db_find(3);       // from=3;
}

void bar() { // version from, to
  spawn db_update(3, 12); // from=2, to=5
  spawn db_find(3);       // from=6
}

void main() {
  spawn foo(); // from=0, to=5;
  spawn bar(); // from=5, to=7;
  spawn db_find(3); from=7
}
```

(a) Task-based code example. The *spawn* keyword creates a task that executes asynchronously.



(b) The task hierarchy generated by the code example.

Figure 2: Pseudo code of a task-based program and the resulting task hierarchy.

is usually impractical. We are currently investigating a dynamic scheme that allows out-of-order spawning, by having each part of the identifier reflect a different nesting level).

Explicit data versioning enables tasks to *spawn* out-of-order rather than only *execute* out-of-order. This is the result of using task identifiers as data versions, and having task identifiers rely on the program's sequential order rather than parallel execution. Note that the parallelism exposed in a task-based program that uses O-structures is a direct function of the ability of the compiler to track and express dependencies between tasks. If the compiler (or programmer) does not / cannot express the dependencies then extra synchronization between tasks is required.

5. Managing data structures

In this section we discuss how programs that manipulate various data structures would use O-structures and additional synchronization to achieve parallel execution.

5.1 Static data structures

Operations on static data structures, e.g. arrays, do not alter their structure but only their contents. If the accesses are statically analyzable then the dependencies between tasks can be encoded in the Release operations to unlock significant parallelism; e.g., a Release of version x can be made directly to the specific version y.

On cases in which dependencies are not statically analyzable, the compiler must introduce some serialization. Consider the creation of a histogram: the compiler cannot predict the order of accesses to each array element, but all accesses to array elements go through a pointer to the array. That pointer could be used as a serialization point, always being released to the next task (and not beyond it). The elements can be acquired before writing, and released to θ after writing. If the pointer is released after the array element is acquired, overall sequential semantics are guaranteed, while accesses to different elements can be done in parallel.

5.2 Dynamic data structures

Operations on dynamic data structures such as lists and trees impose a greater challenge. The connectivity and memory layout of these data structures are determined by dynamic pointers. Mutating operations such as insert and delete not only change the content stored in the nodes, but also modify the node pointers, altering the layout of the data structure itself. Further, elements in these structures are not accessed directly but rather by first traversing a series of other nodes.

The *unipath property* is an algorithmic characteristic of operations, which always follow the same path in order to reach a certain element of a data structure. Traversal of a singly-linked list, for example, has the unipath property: in order to reach a node, all previous nodes starting from the head must always be followed. On the other hand, mutations of balanced trees do not have the unipath property, because rebalancing can cause a revisit of nodes that have already been visited. Operations on data structures like singly-linked list, singly-linked trees and arrays reachable from a single pointer are inherently unipath; such data structures are therefore considered *unipath data structures*. Operations and data structures that do not have the unipath property are considered to be *multipath*.

5.3 Parallelism in unipath operations

Using O-structures allows operations on unipath data structures such as linked list to be executed in parallel, by pipelining subsequent operations. Given a single path to any node in

the list, proper acquiring and releasing guarantees mutations take place in sequential order. Let us consider the series of tasks demonstrated in Figure 3, appending nodes to a list. In order to get to the end of the list, a task must traverse all existing nodes. During traversal, the task first acquires the next node (for read and then write), and then releases the current node to the next task. By having at least one node acquired at any given moment during traversal, each task prevents the following tasks from bypassing it, and appending their nodes before its. Such pipelining allows $O(n)$ operations to run in parallel. Other mutating operations such as deleting or inserting into the middle of the list can be pipelined in a similar fashion, as long as the unipath property is kept.

Operations on trees can also be run in parallel, using a slightly different pipelining technique. Let us consider addition of random numbers to a tree, as demonstrated in Figure 4. The unipath property exists because there is a single path from the root to each node. However, unlike linked list, subsequent operations might not follow the same path. A task that acquires the next node cannot release the current one to the next task, because the next task might never reach that node. Instead, only the root is being released to the next task, because it is guaranteed that the next task will in fact access it. During traversal, all other nodes are released to θ, making them available to any future task. Given each task still acquires at least one node at any given moment, subsequent tasks that traverse the same path cannot bypass it. On the other hand, tasks that traverse different paths can execute in parallel. Pipelining allows $O(d)$ operations to run in parallel, given d is the depth of the tree.

Linked lists Figure 3 illustrate a sequence of insertions to a singly linked list. The operational flow of inserting to a linked list is as follows:

6. Evaluation

6.1 Simulation framework

Our evaluation measures the amount of memory-level parallelism (MLP) that could be obtained when operating on several common data structures. The MLP criteria distills the effect of using O-structures, which effectively provide both versioned memory and synchronization. Computational operations between memory operations, which can freely be executed in parallel, are not considered in the evaluation.

Our infrastructure is trace-based. We first implemented sequential versions of all the algorithms we consider: insertion to the end of a singly linked list, insertion to a binary tree, multiplication of 3 dense matrices and multiplication of 3 sparse matrices. We hand-annotated these codes with task spawn and O-structure Acquire, Release, Read and Write operations. Our approach was a "best a compiler could achieve". The simulator takes these execution traces, and models the tasks executing on a fixed number of processing units. The simulation follows the semantics of task based execution on a system that uses O-structures: a read must stall if the target version could not be acquired; writes are performed immediately and create a new version of the data; tasks are created out-of-order but all dependencies between tasks are faithfully modeled. Our evaluation considers a large number (128) of processing units, although more parallel execution resources and larger data set sizes improve performance. The amount of MLP is obtained by dividing the number of O-structure operations on a given trace by the number of cycles required to complete the parallel execution of the trace (operations are assumed to take 1 cycle to complete). Because these algorithms are implemented sequentially, it is sufficient to assume the "baseline" MLP for a conventional von Neumann execution system is 1.

6.2 Experimental results

Linked lists insertion and matrix multiplications scale nicely, while binary tree insertion scales with the height of the tree.

On the linked list benchmark (Figure 5(a)), insertions are pipelined. This bounds the number of active insertions at any given moment to the number of elements that have previously been inserted. As linked lists grow in size more parallelism is available!

The binary tree insertion (Figure 5(b)) enforces no serialization of tasks that traverse different paths, but the root still serializes all insertions. This bounds the maximal number of tasks actively traversing the tree to the height of the tree, which ranges between N and $\log_2(N)$. Consequently, the inherent amount of MLP grows slowly with respect to the input size. Namely, insertions to trees obtain lower MLP than insertion to lists because although the insertions become active at an identical rate, trees insertions are completed much faster, leaving less time for multiple tasks to run in parallel.

On the dense matrix benchmark (Figure 5(c)), each element of the temporary matrix can be calculated independently. Dependency exists between the calculation of the temporary matrix and the second multiplication, but given the total amount of work, its overhead is negligible.

Sparse matrices (Figure 5(d)) are stored as arrays of linked lists, each list holding only non-zero elements in the row it represents. Multiplying 3 matrices could suffer from (1) the overheads introduced by having to wait for the temporary matrix to be calculated and (2) inserting to short linked lists, which bound the number of active insertion tasks. However, sparse matrices are commonly used for very low percentages of non-zero elements. If the percent of actual insertions is low enough, most of the work is done by tasks that can run in parallel. Consequently, the amount of MLP not only depends on the size of the matrices, but also on the percent of non-zero elements and their locations.

In our experiments, sparse matrices obtained higher MLP than dense matrices. This is only an artifact of using larger input sizes. When larger dense matrices are multiplied the obtained MLP is higher. It is worth noting that matrix multiplication is a highly parallelizable operation, and in our experiments large enough workloads were only bound by the number of simulated processors.

7. Conclusions and Future Work

This paper has introduced O-structures, a novel memory element designed to facilitate parallelism in task-based execution models. We began by describing the complete se-

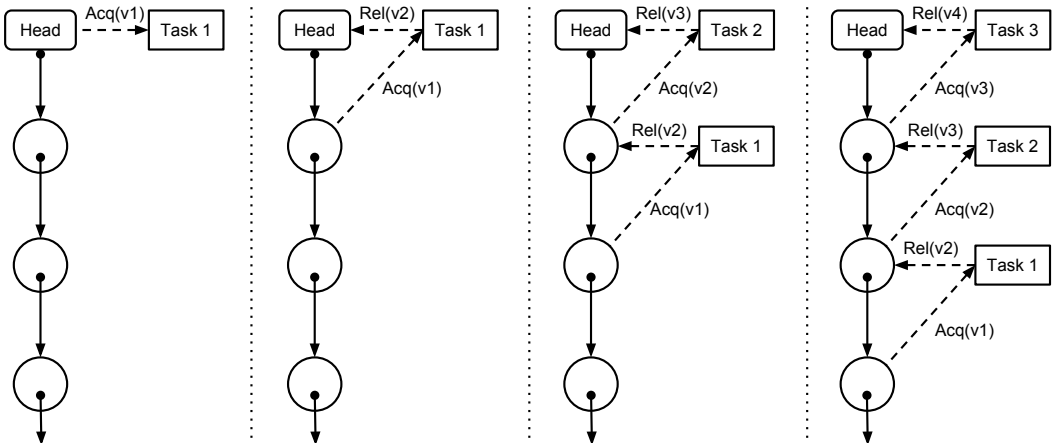


Figure 3: Linked list insertion.

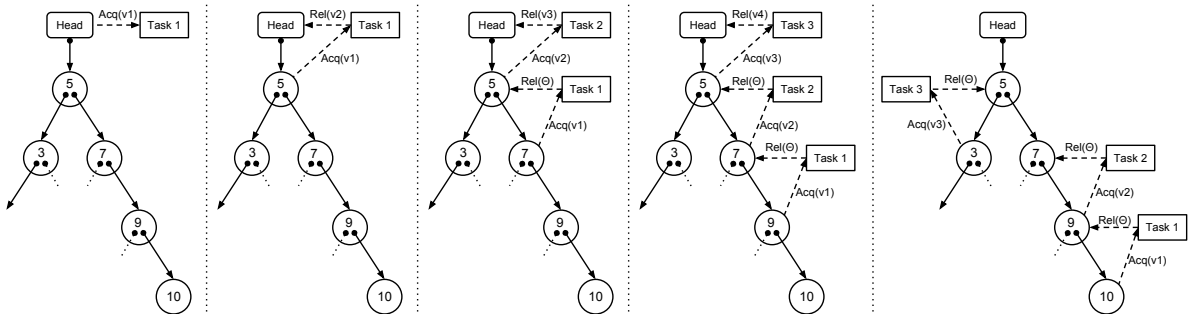


Figure 4: Binary tree insertion.

mantics of O-structures. We then showed how O-structures expose parallelism by removing false output name dependencies in various data structures software developers commonly employ. Finally, we presented results from our execution model simulator that demonstrated significant MLP (over 100X) from sequentially coded implementations of algorithms when O-structures are employed.

At the moment our research group is exploring the microarchitectual implementation of O-structures, compiler and runtime support, and additional semantics for supporting cross-thread synchronization. At the microarchitectural level, caching keeps active O-structures and versions near computation logic. Specialized logic accelerates management of version state. Versions are essentially an extension of the address space, and conceptually by concatenating the version identifier to the address, and then hashing this larger address back down to a smaller table in memory, the various version state can be compactly stored; this is at least the tactic we are presently taking in our design work.

Compiler and runtime support is crucial to the success of O-structures. Benchmarks presented here were hand "compiled". Effective small-scope alias analysis is crucial to performance. Fortunately, alias analysis algorithms work best when focused on small regions of code. At the runtime layer, it's likely adjustments to the memory allocator will be required in order to prevent it from introducing avoidable serialization. Modifications to the runtime system to support synchronization across programmer directed threads are required. Specifically threads must know the version of memory to access in a different programmer directed thread. Exactly how to achieve this without making every synchronization operation a barrier synchronization among tasks within the same programmer directed thread is an open question.

Acknowledgments

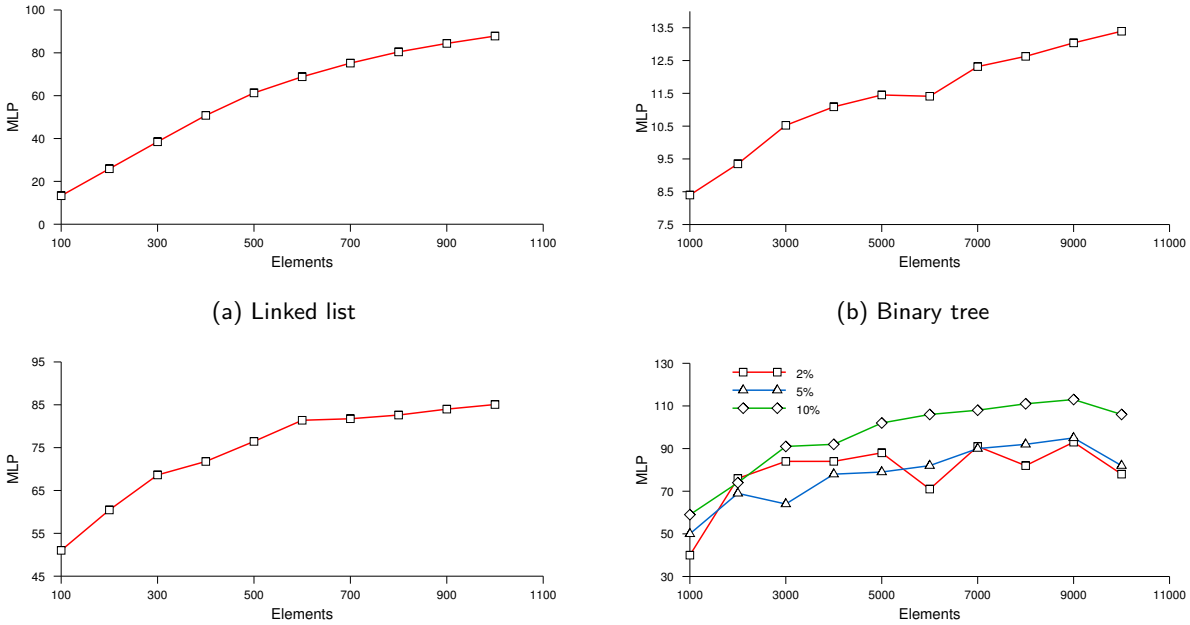
This research was funded in part by the Israel Science Foundation (ISF grant 769/12; equipment grant 1719/12) and by the Intel Collaborative Research Institute for Computational Intelligence (ICRI-CI). Y. Etsion was supported by the Center for Computer Engineering at Technion. In addition this research was funded in part by the National Science Foundation CCF-1335466, Google and the Bershad/Zhu Fellowship.

References

[1] M. D. Allen, S. Sridharan, and G. S. Sohi, "Serialization sets: a dynamic dependence-based parallel execution model," in *PPoPP*, 2008.

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

[3] Arvind, R. S. Nikhil, and K. K. Pingali, "I-structures: data structures for parallel computing," *TOPLAS*, vol. 11, no. 4, Oct. 1989.



(a) Linked list

(b) Binary tree

(c) Dense matrix

(d) Sparse matrix

Figure 5: MLP when using 128 processing cores.

[4] 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.

[5] 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.

[6] P. Bellens, J. Perez, R. Badia, and J. Labarta, "CellSs: a programming model for the Cell BE architecture," *SC*, Nov 2006.

[7] R. Blumofe, M. Frigo, C. Joerg, C. Leiserson, and K. Randall, "DAG-consistent distributed shared memory," in *IPPS*, Apr 1996.

[8] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama, "Impulse: Building a smarter memory controller," in *HPCA*, 1999. [Online]. Available: http://dl.acm.org/citation.cfm?id=520549.822749

[9] D. E. Culler, K. E. Schauser, and T. von Eicken, "Two fundamental limits on dataflow multiprocessing," in *PACT*, 1993.

[10] W. J. Dally, A. Chien, S. Fiske, W. Horwat, J. Keen, M. Larivee, R. Lethin, P. Nuth, S. Wills, P. Carrick, and G. Fyler, "The J-Machine: A fine-grain parallel computer," in *Computing Systems in Engineering*, 1992.

[11] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," in *ISCA*, 1975.

[12] 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.

[13] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *PLDI*, 1998.

[14] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi, "Speculative versioning cache," in *HPCA*, 1998.

[15] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer," in *IEEE Trans. on Computers*, Feb 1982.

[16] G. Gupta, S. Sridharan, and G. S. Sohi, "The road to parallelism leads through sequential programming," in *HotPar*, Jun 2012.

[17] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin, and J. Park, "Mapping irregular applications to DIVA, a PIM-based data-intensive architecture," in *Supercomputing*, 1999.

[18] Y. Kang, M. Huang, S.-M. Yoo, Z. Ge, D. Keen, V. Lam, P. Pattnaik, and J. Torrellas, "FlexRAM: Toward an Advanced Intelligent Memory System," in *Intl. Conf. on Computer Design*, Oct 1999.

[19] C. E. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanović, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick, "Scalable processors in the billion-transistor era: IRAM," *IEEE Computer*, vol. 30, no. 9, 1997.

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

[21] *OpenMP Application Program Interface Version 4.0*, OpenMP Architecture Review Board, Jul 2013.

[22] 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.

[23] E. I. Organick, *A programmer's view of the Intel 432 system*. New York, NY, USA: McGraw-Hill, Inc., 1983.

[24] M. Oskin, F. T. Chong, and T. Sherwood, "Active pages: a computation model for intelligent memory," in *ISCA*, 1998.

[25] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*, Jul 2007.

[26] 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 *Intl. Conf. on Supercomputing (ICS)*, Jun 2005.

[27] M. C. Rinard, D. J. Scales, and M. S. Lam, "Jade: A high-level, machine-independent language for parallel programming," *IEEE Computer*, vol. 26, 1993.

[28] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley, 2010.

[29] C. Segulja and T. Abdelrahman, "Architectural support for synchronization-free deterministic parallel programming," in *HPCA*, 2012.

[30] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar processors," in *ISCA*, 1995.

[31] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "WaveScalar," in *MICRO*, Dec 2003.