Exploiting Locality and NUMA in Scalable Concurrent Libraries

Elad Gidron

Exploiting Locality and NUMA in Scalable Concurrent Libraries

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science

Elad Gidron

Submitted to the Senate of
the Technion — Israel Institute of Technology
Elul 5772 Haifa September 2012

The research thesis was done under the supervision of Prof. Idit Keidar from the Electrical Engineering Department, Technion, in the Computer Science Department, Technion.

I would like to thank my supervisor, Prof. Idit Keidar for guiding me and for showing me how real research is done.
I would also like to thank my college and friend Dmitri Perelman for working with me and for his great help during my research.
Finally, I would like to thank my spouse Ruty, for her support during my research

The generous financial support of the Technion is gratefully acknowledged.

Publications

Parts of this work were published in:

Elad Gidron, Idit Keidar, Dmitri Perelman, and Yonathan Perez. SALSA: scalable and low synchronization NUMA-aware algorithm for producer-consumer pools. In *Proceedinbgs of the 24th ACM symposium on Parallelism in algorithms and architectures*, SPAA '12, 2012

Contents

List of Figures

Abstract

Emerging computer architectures pose many new challenges for software development. First, as the number of computing elements constantly increases, the importance of *scalability* of parallel programs becomes paramount. Second, accessing memory has become the principal bottleneck, while multi-CPU systems are based on NUMA architectures, where memory access from different chips is asymmetric. Therefore, it is instrumental to design software with *local data access*, *cache-friendliness*, and *reduced contention* on shared memory locations, especially across chips. Furthermore, as systems get larger, their behavior becomes less predictable, underscoring the importance of *robust* programs that can overcome unexpected thread stalls.

In our work we focus on two problems:

1. We design and implement a scalable and highly-efficient non-blocking consumer producer task pool, with lightweight synchronization-free operations in the common case. Its data allocation scheme is cache-friendly and highly suitable for NUMA environments. Moreover, our pool is robust in the face of imbalanced loads and unexpected thread stalls.

2. We consider the case of improving metadata locality in word-based STMs. To this end, we evaluate a locality-conscious approach for maintaining versioned locks in TL2. The speedup of the improved algorithm reaches a hundred percent on STAMP benchmarks. We show that this speedup stems from the following factors: 1) improved spacial and temporal locality, 2) reduced false sharing and 3) less false conflicts.

Chapter 1

SALSA: Scalable and Low Synchronization NUMA-aware Algorithm for Producer-Consumer Pools

1.1 Introduction

In this chapter, we focus on one of the fundamental building blocks of highly parallel software, namely a producer-consumer task pool. Specifically, we present a scalable and highly-efficient non-blocking pool, with lightweight synchronization-free operations in the common case. Its data allocation scheme is cache-friendly and highly suitable for NUMA environments. Moreover, our pool is robust in the face of imbalanced loads and unexpected thread stalls.

Our system is composed of two independent logical entities: 1) *SALSA, Scalable and Low Synchronization Algorithm*, a single-consumer pool that exports a stealing operation, and 2) a work stealing framework implementing a management policy that operates multiple SALSA pools.

In order to improve locality and facilitate stealing, SALSA keeps tasks in chunks, organized in per-producer chunk lists. Only the producer mapped to a given list can insert tasks to chunks in this list, which eliminates the need for synchronization among producers.

Though each consumer has its own task pool, inter-consumer synchronization

is required in order to allow stealing. The challenge is to do so without resorting to costly atomic operations (such as CAS or memory fences) upon each task retrieval. We address this challenge via a novel chunk-based stealing algorithm that allows consume operations to be synchronization-free in the common case, when no stealing occurs, which we call the *fast path*. Moreover, SALSA reduces the stealing rate by moving entire chunks of tasks in one steal operation, which requires only two CAS (compare-and-swap) operations.

In order to achieve locality of memory access on a NUMA architecture, SALSA chunks are kept in the consumer's local memory. The management policy matches producers and consumers according to their proximity, which allows most task transfers to occur within a NUMA node.

In many-core machines running multiple applications, system behavior becomes less predictable. Unexpected thread stalls may lead to an asymmetric load on consumers, which may in turn lead to high stealing rates, hampering performance. SALSA employs a novel auto-balancing mechanism that has producers insert tasks to less loaded consumers, and is thus robust to spurious load fluctuations.

We have implemented SALSA in C++, and tested its performance on a 32-core NUMA machine. Our experiments show that the SALSA-based work stealing pool *scales linearly* with the number of threads; it is 20 times faster than other work-stealing alternatives, and shows a significant improvement over state-of-the-art non-FIFO alternatives. SALSA-based pools scale well even in unbalanced scenarios.

This chapter proceeds as follows. Section 1.2 describes related work. We give the system overview in Section 1.4. The model and problem definitions are presented in Section 1.3 he SALSA single-consumer algorithm is described in Section 1.5. We discuss our implementation and experimental results in Section 1.6, and the correctness of our system in Section 1.7. And finally we present our conclusions in Section 1.8.

1.2 Related Work

Task pools. Consumer-producer pools are often implemented as FIFO queues. A widely used state-of-the-art FIFO queue is Micheal and Scott's queue [31]. This queue is implemented by a linked-list with *head* and *tail* references. The put operation adds a new node to the list and then updates the tail reference. This is done

by two CAS operations; one for adding the new node and one for updating the tail reference. The get operation removes a node by moving the head reference to point to the next node. This approach is not scalable under high contention as only one contending operation may succeed.

Moir et al. [32] suggest using elimination to reduce the contention on the queue. Whereby put and get operations can eliminate each other during the back-off after an unsuccessful operation. However, due to the FIFO property, those eliminations can only be done when the queue is empty, making this approach useful only when the queue is almost to empty.

Hoffman et al. [25] try to reduce the contention of the put operation by allowing concurrent put operations to add tasks to the same "basket". This is done by detecting contention on the tail, which is indicated by a failed CAS operation when trying to update the tail. This reduces the contention on the tail, but not on adding the node to the "basket", which still requires a CAS operation. Therefore, this approach, while more efficient than Micheal and Scott's queue, is still not scalable under high contention.

Gidenstam et al. [20] use a similar approach to Micheal and Scott's, but, in order to improve locality and decrease the contention on the head and tail, the data is stored in chunks, and the head and tail points to a chunk rather than single nodes. This allows updating these references only once per-chunk rather than on every operation. However, this solution still requires at least one CAS per operation, rendering it non-scalable under high contention.

A number of previous works have recognized this limitation of FIFO queues, and observed that strict FIFO order is seldom needed in multi-core systems.

Afek et al. [2] implemented a non-FIFO pool using diffraction trees with elimination (ED-pools). An ED-pool is a tree of queues, which contains elimination arrays that help reduce contention. While ED-pools scale better than FIFO based solutions, they do not scale on multi-chip architectures [6].

Basin et al. [7] suggest a wait-free task-pool that allows relaxing FIFO. This pool is more scalable than previous solutions, but, since it still has some ordering (fairness) requirements, there is contention among both producers and consumers.

The closest non-FIFO pool to our work is the Concurrent Bags of Sundell et al. [37], which, like SALSA, uses per-producer chunk lists. This work is optimized for the case that the same threads are both consumers and producers, and typically consume from themselves, while SALSA improves the performance of such a task pool in NUMA environments where producers and consumers are sep-

arate threads. Unlike our pool, the Concurrent Bags algorithm uses strong atomic operations upon each consume. In addition, steals are performed in the granularity of single tasks and not whole chunks as in SALSA. Overall, their throughput does not scale linearly with the number of participating threads, as shown in [37] and in Section 1.6 of this chapter.

To the best of our knowledge, all previous solutions use strong atomic operations (like CAS), at least in every consume operation. Moreover, most of them [2, 3, 7] do not partition the pool among processors, and therefore do not achieve good locality and cache-friendliness, which has been shown to limit their scalability on NUMA systems [6].

Techniques. Variations of techniques we employ were previously used in various contexts. Work stealing [9] is a standard way to reduce contention by using individual per-consumer pools, where tasks may be stolen from one pool to another. We improve the efficiency of stealing by transferring a chunk of tasks upon every steal operation. Hendler et al. [23] have proposed stealing of multiple items by copying a range of tasks from one dequeue to another, but this approach requires costly CAS operations on the fast-path and introduces non-negligible overhead for item copying. In contrast, our approach of chunk-based stealing coincides with our synchronization-free fast-path, and steals whole chunks in O(1) steps. Furthermore, our use of page-size chunks allows for data migration in NUMA architectures to improve locality, as done in [8].

The principle of keeping NUMA-local data structures was previously used by Dice et al. for constructing scalable NUMA locks [15]. Similarly to their work, our algorithm's data allocation scheme is designed to reduce inter-chip communication.

The concept of a synchronization-free fast-path previously appeared in works on scheduling queues, e.g., [4, 22]. However, these works assume that the same process is both the producer and the consumer, and hence the synchronization-free fast-path is actually used only when a process transfers data to *itself*. Moreover, those works assume a sequentially consistent shared-memory multiprocessor system, which requires insertion of some memory barrier instructions to the code when implemented on machine providing a weaker memory model [5]. On the other hand, our pool is synchronization-free even when tasks are transfered among multiple threads; our synchronization-free fast-path is used also when multiple producers produce data for a single consumer. We do not know of any other work that

supports synchronization-free data transfer among different threads.

The idea of organizing data in chunks to preserve locality in dynamically-sized data structures was previously used in [10, 20, 22, 37]. SALSA extends on the idea of chunk-based data structures by using chunks also for efficient stealing.

1.3 Model and Problem Definitions

The problem we solve in this chapter is implementing a lock-free linearizable task-pool. In Section 1.3.1 we describe the model and runtime environment. Then, in Section 1.3.2, we define the linearizability criterion for concurrent data structures. In Section 1.3.3, we introduce a sequential specification for task pools. Finally, in Section 1.3.4, define our progress guarantee, namely lock-freedom.

1.3.1 Implementation Environment

We consider a shared memory environment where execution threads have a shared heap, shared read only code, and separate stack memory spaces. The scheduler can suspend a thread, for an arbitrary duration of time, at any moment after termination of a basic processor instruction (read, write, CAS). Threads cannot be suspended in the middle of a basic instruction. In modern architectures read and write operations may be reordered unless explicitly using a fence operation. However, in our model we assume sequential execution of instruction per-thread. The reordering problems are solved by using implicit fences when using CAS, or by the technique explained in 1.6.1.

1.3.2 Concurrent Objects, Linearizability

Formally, a task pool is a *concurrent object* [24], which resides in a memory shared among multiple threads. As a concurrent object, it has some state and supports a set of operations. Multiple threads can simultaneously perform operations on the same object. Such operations may update the state of the object. Operations take time and have a moment of *invocation* and a moment of *response*. When threads concurrently perform operations on concurrent objects, they generate a *history* [24], which is an ordered list of invocation and response events of concurrent object operations. The order of events is according to the time line in which they occurred. An operation invocation event is represented by the record $O.method_T(args)$, where O is the concurrent object, method is the invoked operation, args are the

invocation arguments and T is the thread that started the invocation. An operation response event is represented by the record $O.method_T(args)$ returns $result$, where $result$ is the operation's result. In a given history, we say that a response matches a prior invocation if it has the same object O and thread T, and no other events of T on object O appear between them. A *sequential history* is a history that has the following properties: 1) the first event in the history is an invocation; 2) each invocation, except possibly the last, is immediately followed by a matching response.

A sequential specification defines which sequential histories of an object are legal.

For defining the correctness of concurrent objects we consider the following definitions. An invocation is pending in history H if no matching response follows the invocation. An extension of *history* H is a history constructed by appending zero or more responses matching the pending invocations of H. $Complete(H)$ is the sub-sequence of H created by removing all pending invocations of H. $H|T$ is a history consisting of exactly the events of thread T in history H. Two histories H and H' are equivalent if for each thread T, $H|T = H'|T$.

Given a sequential specification of a concurrent object, the *linearizability* [24] correctness criterion is defined as follows: A history H is *linearizable* if it has an extension H' and there is a sequential history S such that:

1. S is legal according to the sequential specification of the object.
2. Complete(H') is equivalent to S.
3. If method response m' precedes method invocation m in H, then the same is true in S.

Concurrent objects that have only linearizable histories are called *linearizable* or *atomic*. Intuitively, a concurrent object is linearizable if it requires each concurrent run of its method calls to be equivalent in some sense to a correct serial run.

1.3.3 Task Pool Sequential Specification

A task pool supports $put(T)$ and $get()$ returns T operations, where T is a task or $\perp$.

We assume that tasks inserted into the pool are unique. That is, if $put(T)$ and $put(T')$ are two different invocations on a task pool, then $T \neq T'$. This assumption

is made to simplify the definitions, and could be easily enforced in practice by tagging tasks with process ids and sequence numbers. The sequential specification of a task pool is as follows:

$put(T)$ operation adds task T to the pool. $get()$ returns and removes a task T from the pool or returns $\perp$ if the pool is empty.

1.3.4 Lock-freedom

Threads may invoke a concurrent object's operations simultaneously. A concurrent object implementation is *lock-free* if there is guaranteed system-wide progress, i.e., at least one thread always makes progress in its operation execution, regardless of the execution speeds or failures of other threads. In this chapter, we implement a *lock-free* shared object.

1.4 System Overview

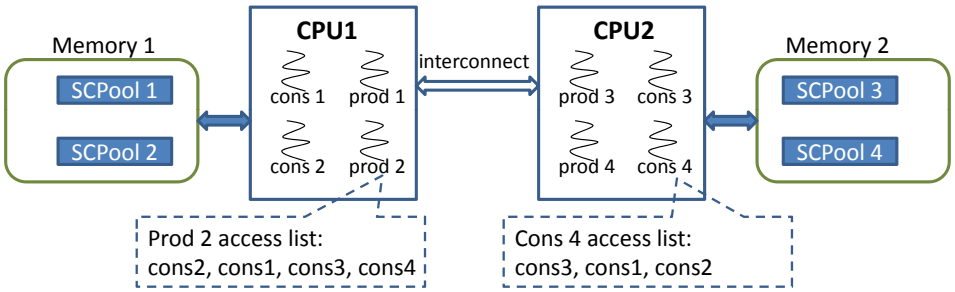


Figure 1.1: Producer-consumer framework overview. In this example, there are two processors connected to two memory banks (NUMA architecture). Two producers and two consumers running on each processor, and the data of each consumer is allocated at the closest physical memory. A producer (consumer) has a sorted access list of consumers for task insertion (respectively stealing).

In the current section we present our framework for scalable and NUMA-aware producer-consumer data exchange. Our system follows the principle of separating mechanism and policy. We therefore consider two independent logical entities:

1. *A single consumer pool (SCPool)* mechanism manages the tasks arriving to a given consumer and allows tasks stealing by other consumers.

2. A management policy operates SCPools: it routes producer requests to the appropriate consumers and initiates stealing between the pools. This way,

the policy controls the system's behavior according to considerations of load-distribution, throughput, fairness, locality, etc. We are especially interested in a management policy suitable for NUMA architectures (see Figure 1.1), where each CPU has its own memory, and memories of other CPUs are accessed over an interconnect. As a high rate of remote memory accesses can decrease the performance, it is desirable for the SCPool of a consumer to reside close to its own CPU.

Algorithm 1 API for a Single Consumer Pool with stealing support.

1: boolean: produce(Task, SCPool) ▷ Tries to insert the task to the pool, returns false if no space is available.
2: void: produceForce(Task, SCPool) ▷ Insert the task to the pool, expanding the pool if necessary.
3: {Task ∪⊥}: consume() ▷ Retrieve a task from the pool, returns ⊥ if no tasks in the pool are detected.
4: {Task ∪⊥}: steal(SCPool from) ▷ Try to steal a number of tasks from the given pool and move them to the current pool. Return some stolen task or ⊥.
5: boolean: isEmpty() ▷ Returns true iff the SCPool contains tasks
6: void: setIndicator(SCPool p, int consumerId) ▷ sets indicator in pool *p* of consumer *consumerId*
7: boolean: checkIndicator(SCPool p, int consumerId) ▷ returns the state of the indicator in pool *p* of consumer *consumerId*

SCPool abstraction. The SCPool API provides the abstraction of a single consumer task pool with stealing support, see Algorithm 1. A producer invokes two operations: **produce()**, which attempts to insert a task to the given pool and fails if the pool is full, and **produceForce()**, which always succeeds by expanding the pool on demand. There are also two ways to retrieve a task from the pool: the owner of the pool (only) can call the **consume()** function; while any other thread can invoke **steal()**, which tries to transfer a number of tasks between two pools and return one of the stolen tasks. The other function are used for checking emptiness and will be explained in 1.5.5.

A straightforward way to implement the above API is to use a dynamic-size multi-producer multi-consumer FIFO queue (e.g., Michael-Scott queue [31]). In this case, **produce()** enqueues a new task, while **consume()** and **steal()** dequeue a task. In the next section we present SALSA, a much more efficient SCPool.

Algorithm 2 Work stealing framework pseudo-code.

```
 8: Local variables:
 9:     SCPool myPool ▷ The consumer's pool
10:     SCPool[] accessList ▷ The consumer's or producer's access list

11: Function get():
12:     while(true)
13:         ▷ First try to get a task from the local pool
14:         t ← myPool.consume()
15:         if (t ≠ ⊥) return t
16:         ▷ Failed to get a task from the local pool – steal
17:         foreach SCPool p in accessList in order do:
18:             t ← p.steal()
19:             if (t ≠ ⊥) return t
20:         ▷ No tasks found – validate emptiness
21:         if (checkEmpty()) return ⊥

22: Function put(Task t):
23:     ▷ Produce to the pools by the order of the access list
24:     foreach SCPool p in accessList in order do:
25:         if (p.produce(t)) return
26:     firstp ← the first entry in accessList
27:     ▷ If all pools are full, expand the closest pool
28:     produceForce(t,firstp)
29:     return

30: Function checkEmpty():
31:     for i in {1..|consumers|} do:
32:         foreach SCPool p do:
33:             if (i = 1) p.setIndicator(myId)
34:             if (!p.isEmpty()) return false
35:             if (!p.checkIndicator(myId)) return false
36:     return true
```

Management policy. A management policy defines the way in which: 1) a producer chooses an SCPool for task insertion; and 2) a consumer decides when to retrieve a task from its own pool or steal from other pools. Note that the policy is independent of the underlying SCPool implementation. We believe that the policy is a subject for engineering optimizations, based on specific workloads and demands.

In the current work, we present a NUMA-aware policy. If the individual SCPools themselves are lock-free, then our policy preserves lock-freedom at the system level. Our policy is as follows:

- **Access lists.** Each thread in the system (producer or consumer) is provided with an *access list*, an ordered list of all the consumers in the system, sorted according to their distance from that thread (see Figure 1.1). Intuitively, our intention is to have a producer mostly interact with the closest consumer, while stealing mainly happens inside the same processor node.

- **Producer's policy.** The producer policy is implemented in the **put**() function in Algorithm 2. The operation first calls the **produce()** of the first SCPool in its access list. Note that this operation might fail if the pool is full, (which can be seen as evidence of that the corresponding consumer is overloaded). In this case, the producer tries to insert the task into other pools, in the order defined by its access list. If all insertions fail, the producer invokes **produceForce()** on the closest SCPool, which always succeeds (expanding the pool

if needed).

- **Consumer's policy.** The consumer policy is implemented in the **get**() function in Algorithm 2. A consumer takes tasks from its own SCPool. If its SCPool is empty, then the consumer tries to steal tasks from other pools in the order defined by its access list. The **checkEmpty**() operation handles the issue of when a consumer gives up and returns $\perp$. This is a subtle issue, and we discuss it in Section 1.5.5. Stealing serves two purposes: first, it is important for distributing the load among all available consumers. Second, it ensures that tasks are not lost in case they are inserted into the SCPool of a crashed (or very slow) consumer.

1.5 Algorithm Description

In the current section we present the SALSA SCPool. We first show the data structures of SALSA in Section 1.5.1, and then present the basic algorithm without stealing support in Section 1.5.2. The stealing procedure is described in Section 1.5.3, finally, the role of chunk pools is presented in Section 1.5.4. For the simplicity of presentation, in this section we assume that the the memory accesses satisfy sequential consistency [27], we describe the ways to solve memory reordering issues in Section 1.6.1.

1.5.1 SALSA Structure

Algorithm 3 SALSA implementation of SCPool: Data Structures.

```
37: Chunk type
38:     Task[CHUNK_SIZE] tasks
39:     int owner ▷ owner's consumer id
40: Node type
41:     Chunk c; initially ⊥
42:     int idx; initially -1
43:     Node next;
44: SALSA per consumer data structure:
45:     int consumerId
46:     List⟨Node⟩[] chunkLists ▷ one list per producer + extra list for stealing (every list is single-writer multi-reader)
47:     Queue⟨Chunk⟩ chunkPool ▷ pool of spare chunks
48:     Node currentNode, initially ⊥ ▷ current node to work with
```

The SALSA data structure of a consumer c_i is described in Algorithm 3 and partially depicted in Figure 1.2. The tasks inserted to SALSA are kept in chunks, which are organized in per-producer chunk lists. Only the producer mapped to a given list can insert a task to any chunk in that list. Every chunk is owned by a

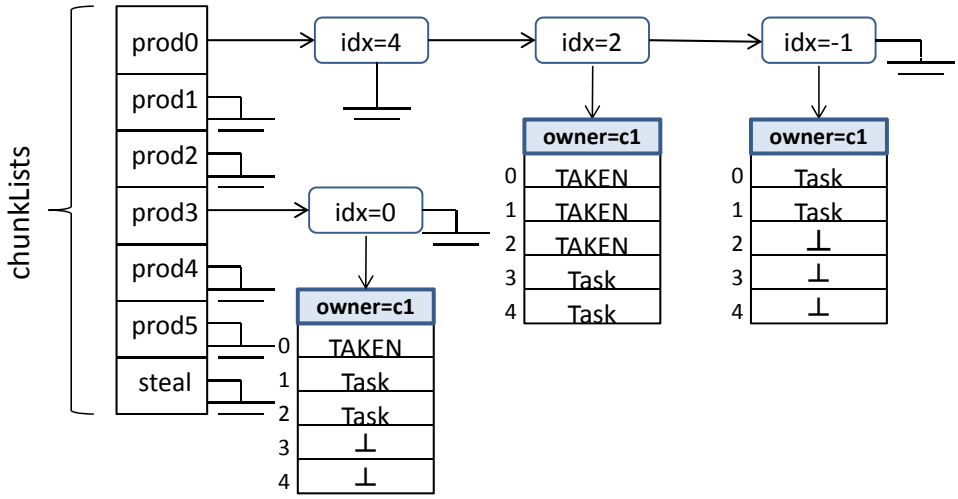


Figure 1.2: Chunk lists in SALSA single consumer pool implementation. Tasks are kept in chunks, which are organized in per-producer lists; an additional list is reserved for stealing. Each list can be modified by the corresponding producer only. The only process that is allowed to retrieve tasks from a chunk is the owner of that chunk (defined by the ownership flag). A Node's index corresponds to the latest task taken from the chunk or the task that is about to be taken by the current chunk owner.

single consumer whose id is kept in the *owner* field of the chunk. The owner is the only process that is allowed to take tasks from the chunk; if another process wants to take a task from the chunk, it should first steal the chunk and change its ownership. A task entry in a chunk is used at most once. Its value is $\perp$ before the task is inserted, and TAKEN after it has been consumed.

The per-producer chunk lists are kept in the array *chunkLists* (see Figure 1.2), where *chunkLists[j]* keeps a list of chunks with tasks inserted by producer p_j. In addition, the array has a special entry *chunkLists[steal]*, holding chunks stolen by c_i. Every list has a single writer who can modify the list structure (add or remove nodes): *chunkLists[j]*'s modifier is the producer p_j, while *chunkLists[steal]*'s modifer is the SCPool's owner. The nodes of the used chunks are lazily reclaimed and removed by the list's owner. For brevity, we omit the linked list manipulation functions from the pseudo-code below. Our single-writer lists can be implemented without synchronization primitives, similarly to the single-writer linked-list in [30]. In addition to holding the chunk, a node keeps the index of the latest taken task in that chunk, this index is then used for chunk stealing as we show in Section 1.5.3.

Safe memory reclamation is provided by using hazard pointers [30] both for

nodes and for chunks. The free (reclaimed) chunks in SALSA are kept at per-consumer *chunkPools* implemented by lock-free Michael-Scott queues [31]. As we show in Section 1.5.4, the chunk pools serve two purposes: 1) efficient memory reuse and 2) producer-based load balancing.

1.5.2 Basic Algorithm

SALSA producer

Algorithm 4 SALSA implementation of SCPool: Producer Functions.

```
49: Producer local variables:
50:     int producerId
51:     Chunk chunk; initially ⊥ ▷ the chunk to insert to
52:     int prodIdx; initially 0 ▷ the prefix of inserted tasks

53: Function produce(Task t):
54:     return insert(t, this, false)

55: Function insert(Task t, SCPool scPool, bool force):
56:     if (chunk = ⊥) then ▷ allocate new chunk
57:         if (getChunk(scPool, force) = false) then return false
58:     chunk.tasks[prodIdx] ← t; prodIdx++
59:     if(prodIdx = CHUNK_SIZE) then
60:         chunk ← ⊥ ▷ the chunk is full
61:     return true

62: Function produceForce(Task t):
63:     insert(t, this, true)

64: Function getChunk(SALSA scPool, bool force)
65:     newChunk ← dequeue chunk from scPool.chunkPool
66:     if (chunk = ⊥) ▷ no available chunks in this pool
67:         if (force = false) then return false
68:         newChunk ← allocate a new chunk
69:     newChunk.owner ← scPool.consumerId
70:     node ← new node with idx = −1 and c = newChunk
71:     scPool.chunkLists[producerId].append(node)

72:     chunk ← newChunk; prodIdx ← 0
73:     return true
```

The description of SALSA producer functions is presented in Algorithm 4. The insertion of a new task consists of two stages: 1) finding a chunk for task insertion (if necessary), and 2) adding a task to the chunk.

Finding a chunk The chunk for task insertions is kept in the local producer variable *chunk* (line 51 in Algorithm 4). Once a producer starts working with a chunk c, it continues inserting tasks to c until c is full – the producer is oblivious to chunk stealing. If the *chunk*'s value is $\perp$, then the producer should start a new chunk (function *getChunk*). In this case, it tries to retrieve a chunk from the chunk pool and to append it to the appropriate chunk list. If the chunk pool is empty then the producer either returns $\perp$ (if force=false), or allocates a new chunk by itself (otherwise) (lines 66–68).

Inserting a task to the chunk As previously described in Section 1.5.1, different producers insert tasks to different chunks, which removes the need for synchronization among producers. The producer local variable *prodIdx* indicates the next free slot in the chunk. All that is left for the insertion function to do, is to put a task in that slot and to increment *prodIdx* (line 58). Once the index reaches the maximal value, the *chunk* variable is set to $\perp$, indicating that the next insertion operation should start a new chunk.

SALSA consumer without stealing

The consumer's algorithm without stealing is given in the left column of Algorithm 5. The consumer first finds a nonempty chunk it owns and then invokes **takeTask()** to retrieve a task.

Unlike producers, which have exclusive access to insertions in a given chunk, a consumer must take into account the possibility of stealing. Therefore, it should notify other processes which task it is about to take.

To this end, each node in the chunk list keeps an index of the taken prefix of its chunk in the *idx* variable, which is initialized to -1. A consumer that wants to take a task T, first increments the index, then checks the chunk's ownership, and finally changes the chunk entry from T to *TAKEN* (lines 90–92). By doing so, a consumer guarantees that *idx* always points to the last taken task or to a task that is about to be taken. Hence, a thread that is stealing a chunk from a node with $idx = i$ can assume that the tasks in the range $[0 \ldots i)$ have already been taken. The logic for dealing with stolen chunks is described in the next section.

1.5.3 Stealing

The stealing algorithm is given in the function **steal()** in Algorithm 5. We refer to the stealing consumer as c_s, the victim process whose chunk is being stolen as c_v, and the stolen chunk as C.

The idea is to turn c_s to the exclusive owner of C, so that c_s will be able to take tasks from the chunk without synchronization. In order to do that, c_s first adds the chunk to its list (line 115) then changes the ownership of C from c_v to c_s using CAS (line 116) and removes the chunk from c_v's list (line 132). Once c_v notices the change in the ownership it can take at most one more task from C (lines 95–98) after failing the second check of ownership in line 91 having passed the one in line 88.

Algorithm 5 SALSA implementation of SCPool: Consumer Functions.

```
74: Function consume():
75:     if (currentNode ≠ ⊥) then ▷ common case
76:         t ← takeTask(currentNode)
77:         if (t ≠ ⊥) then return t
78:     foreach Node n in ChunkLists do: ▷ fair traversal of chunkLists
79:         if (n.c ≠ ⊥ ∧ n.c.owner = consumerId) then
80:             t ← takeTask(n)
81:             if (t ≠ ⊥) then currentNode ← n; return t
82:     currentNode ← ⊥; return ⊥

83: Function takeTask(Node n):
84:     chunk ← n.c
85:     if (chunk = ⊥) then return ⊥ ▷ this chunk has been stolen
86:     task ← chunk.tasks[n.idx + 1]
87:     if (task = ⊥) then return ⊥ ▷ no inserted tasks
88:     if (chunk.owner ≠ consumerId)
89:         return ⊥
90:     n.idx++ ▷ tell the world you're going to take a task from idx
91:     if (chunk.owner = consumerId) then ▷ common case
92:         chunk.tasks[n.idx] ← TAKEN
93:         checkLast(n)
94:         return task
        ▷ the chunk has been stolen, CAS the last task and go away
95:     success ← (task ≠ TAKEN ∧
            CAS(chunk.tasks[n.idx], task, TAKEN))
96:     if(success) then checkLast(n)
97:     currentNode ← ⊥
98:     return (success) ? task : ⊥

99: Function checkLast(Nconsumerode n):
100:    if(n.idx + 1 = CHUNK_SIZE) then ▷ finished the chunk
101:        n.c ← ⊥; return chunk to chunkPool
102:        currentNode ← ⊥

103: Function isEmpty():
104:    foreach Node n in chunkLists do:
105:        if (n.c has tasks in slots greater than n.idx)
106:            return true
107:        return false

108: Function steal(SCPool p):
109:    prevNode ← a node holding tasks, whose owner is p, from some list in p's pool ▷ different policies possible
110:    if (prevNode = ⊥) return ⊥ ▷ No Chunk found
111:    c ← prevNode.c; if (c = ⊥) then return ⊥
112:    prevIdx ← prevNode.idx
113:    if (prevIdx+1 = CHUNK_SIZE ∨ c.tasks[prevIdx+1] = ⊥)
114:        return ⊥
115:    chunkLists[steal].append(prevNode) ▷ make it stealable from my list
116:    if (CAS(c.owner, p.consumerId, consumerId) = false)
117:        chunkLists[steal].remove(prevNode)
118:        return ⊥ ▷ failed to steal
119:    idx ← prevNode.idx
120:    if (idx+1 = CHUNK_SIZE) ▷ Chunk is empty
121:        chunkLists[steal].remove(prevNode)
122:        return ⊥
123:    task ← c.tasks[idx+1]
124:    if (task ≠ ⊥) ▷ Found task to take
125:        if (c.owner ≠ consumerId ∧ idx ≠ prevIdx)
126:            chunkLists[steal].remove(prevNode)
127:            return ⊥
128:        idx++
129:    newNode ← copy of prevNode
130:    newNode.idx = idx
131:    replace prevNode with newNode in chunkLists[steal]
132:    prevNode.c ← ⊥ ▷ remove chunk from consumer's list
        ▷ done stealing the chunk, take one task from it
133:    if (task = ⊥) then return ⊥ ▷ still no task at idx
134:    if (task = TAKEN ∨
            !CAS(c.tasks[idx], task, TAKEN)) then
135:        task ← ⊥
136:    checkLast(newNode)
137:    if (c.owner = consumerId) currentNode ← newNode
138:    return task
```

When the **steal()** operation of c_s occurs simultaneously with the **takeTask()** operation of c_v, both c_s and c_v might try to retrieve the same task. We now explain why this might happen. Recall that c_v notifies potential stealers of the task it is

about to take by incrementing the *idx* value in C's node (line 90). This value is copied by c_s in line 129 when creating a copy of C's node for its steal list.

Consider, for example, a scenario in which the idx is incremented by c_v from 10 to 11. If c_v checks C's ownership before it is changed by c_s, then c_v takes the task at index 11 *without synchronization* (line 92). Therefore, c_s cannot be allowed to take the task pointed by *idx* at all. Hence, c_v has to take the task at index 11 even if it does observe the ownership change. After stealing the chunk, c_s will eventually try to take the task pointed by $idx + 1$. However, if c_s copies the node before idx is incremented by c_v, c_s might think that the value of $idx + 1$ is 11. In this case, both c_s and c_v will try to retrieve the task at index 11. To ensure that the task is not retrieved twice, both functions invoke CAS in order to retrieve this task (line 134 for c_s, line 95 for c_v).

The above schematic algorithm works correctly as long as the stealing consumer can observe the node with the updated index value. This might not be the case in case the same chunk is concurrently stolen by another consumer, rendering the *idx* of the original node obsolete. In order to prevent this situation, stealing a chunk from the pool of consumer c_v is allowed only if c_v is the owner of this chunk (line 116). This approach is prone to the ABA problem: consider a scenario where consumer c_a is trying to steal from c_b, but before the execution of the CAS in line 116, the chunk is stolen by c_c and then stolen back by c_b. In this case, c_a's CAS succeeds but c_a has an old value of idx. To prevent this ABA problem, the owner field contains a tag, which is incremented on every CAS operation. For brevity, tags are omitted from the pseudo-code.

A naïve way for c_s to steal the chunk from c_v would be first to change the ownership and then to move the chunk to the steal list. However, this approach may cause the chunk to disappear when c_s stalls, because the chunk is not yet accessible via the lists of c_s and yet c_s is its owner. Therefore, SALSA first adds the original node to the steal list of c_s, then changes the ownership, and only then replaces the original node with a new one (lines 115–132).

An additional problem may occur if c_s steals a chunk that does not contain tasks. This may happen if the chunk is emptied after c_s chooses it in line 109. In this case, c_s may notice that the chunk does not contain a task and return $\perp$ in line 133. However, another task may be added later and then taken by c_v, which may have already started taking a task before the chunk was stolen. In this case, c_v will take this task using a CAS operation, while c_s may try to take the same task later without using a CAS operation, and therefore the task may be taken twice.

To avoid this problem, we make sure that if a chunk is stolen, c_v will not take a task that c_s might have missed because it was added after c_s tried to read it. This is done by adding an ownership check after c_v reads the task on line 86 and before committing to take it by incrementing idx in line 90. This makes sure that c_v can only take tasks that existed before the chunk was stolen. For the same reason, the ownership check is added in line 125. In this case however c_v also checks if the idx has changed since before it changed ownership. This is done by comparing the idx read before the ownership change in line 112 to the idx read after the ownership change in line 119. If the idx hasn't changed, it means that c_s is guaranteed to see the task pointed by idx, because due to the check in line 113 we know that task existed before c_v changed ownership, and therefore existed before c_s changed ownership. In this case c_v may safely increase idx and take the task. Note that returning the task is necessary to avoid livelock.

Another issue we need to address is making sure that the idx value in nodes pointing to a given chunk increases monotonically. To this end, we make sure that when c_s creates a new node, this node's idx is greater than or equal to the idx of c_v's node. As noted before, c_v may increase the idx at most once after its chunk is stolen. Also, thanks to the ownerships checks that are done after the task was read and before the idx is incremented, we know that the idx field of c_v increases only if there is a task in the next slot after the ownership change. To ensure that idx does not decrease in this case, c_s sets the idx of the new node to be the idx of c_v plus one if the next task is not $\perp$ (line 128).

1.5.4 Chunk Pools

As described in Section 1.5.1, each consumer keeps a pool of free chunks. When a producer needs a new chunk for adding a task to consumer c_i, it tries to get a chunk from c_i's chunk pool – if no free chunks are available, the **produce()** operation fails.

As described in Section 1.4, our system-wide policy defines that if an insertion operation fails, then the producer tries to insert a task to other pools. Thus, the producer avoids adding tasks to overloaded consumers, which in turn decreases the amount of costly steal operations. We further refer to this technique as producer-based balancing.

Another SALSA property is that a chunk is returned to the pool of a consumer that retrieves the latest task of this chunk. Therefore, the size of the chunk pool

of consumer c_i is proportional to the rate of c_i's task consumption. This property is especially appealing for heterogeneous systems – a faster consumer c_i, (e.g., one running on a stronger or less loaded core), will have a larger chunk pool, and so more **produce()** operations will insert tasks to c_i, automatically balancing the overall system load.

1.5.5 Checking Emptiness

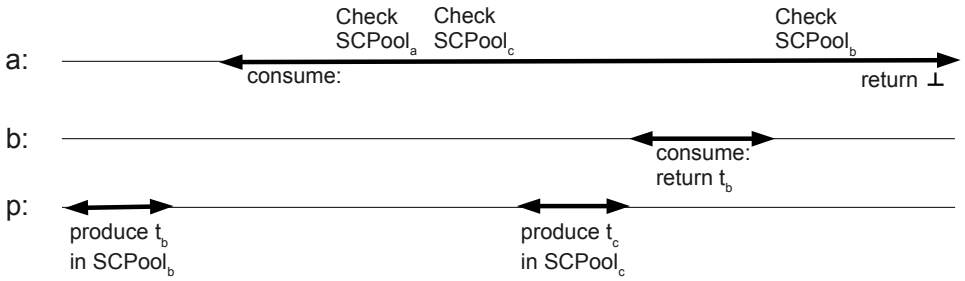


Figure 1.3: An example where a single traversal may violate linearizability: consumer a is trying to get a task. It fails to take a task from its own pool, and starts looking for chunks to steal in other pools. At this time there is a single non-empty chunk in the system, which is in b's pool; a checks c's pool and finds it empty. At this point, a producer adds a task to c's pool and then b takes the last task from its pool before a checks it. Thus, a finds b's pool empty, and returns $\perp$. There is no way to linearize this execution, because throughout the execution of a's operation, the system contains at least one task.

For our system to be linearizable, we must ensure that it returns $\perp$ only if it is empty (i.e., contains no tasks) at some point during the **get**() operation. We describe a policy for doing so in a lock-free manner.

Let us examine why a naïve approach, of simply traversing all task pools and returning $\perp$ if no task is found, violates correctness. First, a consumer might "miss" one task added during its traversal, and another removed during the same traversal, as illustrated in Figure 3. In this case, a single traversal would have returned $\perp$ although the pool was not empty at any point during the **get**() operation. Second, a consumer may miss a task that is moved from one pool to another due to stealing. In order to identify these two cases, we add to each pool a special *emptyIndicator*, a bit array with a bit per-consumer, which is cleared every time the pool *may* become empty. In SALSA, this occurs when the last task in a chunk is taken or when a chunk is stolen. In addition, we implement a new function, **checkEmpty**(), which is called by the framework whenever a consumer fails to retrieve tasks from its

pool and all other pools. This function returns true only if there is a time during its execution when there are no tasks in the system. If **checkEmpty()** returns false, the consumer simply restarts its operation.

Denote by n the number of consumers in the system. The **checkEmpty()** function works as follows: the consumer traverses all SCPools, to make sure that no tasks are present. After checking a pool for the first time, the consumer sets its bit in *emptyIndicator* using CAS. The consumer repeats this traversal n times, where in all traversals, it checks that its bit in *emptyIndicator* is set, i.e., that no chunks were emptied or removed during the traversal. The n traversals are needed in order to account for the case that other consumers have already stolen or removed tasks, but did not yet update *emptyIndicator*, and thus their operations were not detected by the consumer. Since up to $n-1$ pending operations by other consumers may empty pools before any *emptyIndicator* changes, it is guaranteed that among n traversals in which no chunks were seen and the *emptyIndicator* did not change, there is one during which the system indeed contains no tasks, and therefore it is safe to return $\perp$. This method is similar to the one used in Concurrent Bags [37].

Algorithm 6 SALSA extensions for supporting checkEmpty()

```
139: Per consumer local:
140:     boolean[] emptyIndicator ▷ one entry per consumer

    ▷ replacement for the checkLast() function
141: Function checkLast(Node n, Task next):
142:     if(n.idx + 1 = CHUNK_SIZE) then ▷ finished the chunk
143:         n.c ← ⊥; return chunk to chunkPool
144:         currentNode ← ⊥
145:         clearIndicator()
146:     if(next = ⊥) then ▷ took last task
147:         clearIndicator()

148: Function clearIndicator():
149:     foreach(boolean b in emptyIndicator) do:
150:         b ← false

151: Function setIndicator(SCPool p, int consumerId):
152:     emptyIndicator[consumerId] ← true

153: Function checkIndicator(SCPool p, int consumerId):
154:     return emptyIndicator[consumerId]
```

We now describe the extensions to the SALSA pool which are needed so that **checkEmpty**() will work. Specifically, we need to make sure that operations that may cause a pool to become empty will clear *emptyIndicator*.

We note that a pool may become empty in two cases: (1) When a chunk is stolen from a pool and this is the only chunk that contains tasks, and (2) when a task is taken and that was the last task in the pool.

We alter the consumer code so it will clear it in those cases:

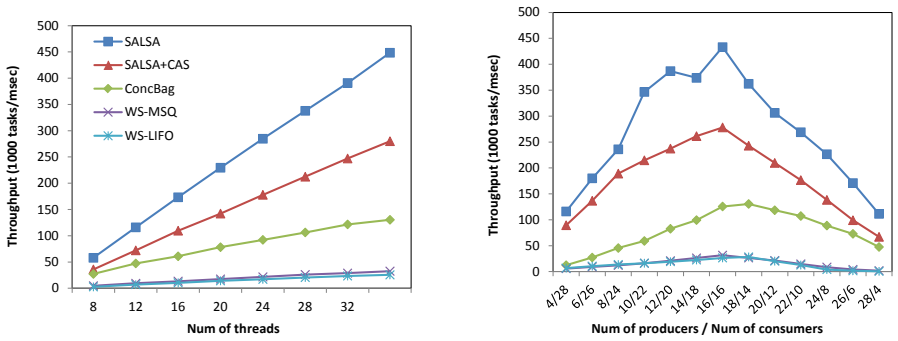
1. In case of a successful steal - the consumer clear the indicator before line 119.

2. If the task returned may be the last task in the chunk, the consumer clears *emptyIndicator* in the **checkLast()** function. The updated function is described in Algorithm 6.

In the second case, the consumer checks that this is the last task by reading the next slot before changing the current slot to TAKEN, and then checking if the next slot contained $\perp$.

1.6 Implementation and Evaluation

In this section we evaluate the performance of our work-stealing framework built on SALSA pools. We first present the implementation details on dealing with memory reordering issues in Section 1.6.1. The experiment setup is described in Section 1.6.2, we show the overall system performance in Section 1.6.3, study the influence of various SALSA techniques in Section 1.6.4 and check the impact of memory placement and thread scheduling in Section 1.6.5.



(a) System throughput – N producers, N consumers. (b) System throughput – variable producers-consumers ratio.

Figure 1.4: System throughput for various ratios of producers and consumers. SALSA scales linearly with the number of threads – in the $16/16$ workload, it is $\times 20$ faster than WS-MSQ and WS-LIFO, and $\times 3.5$ faster than Concurrent Bags. In tests with equal numbers of producers and consumers, the differences among work-stealing alternatives are mainly explained by the consume operation efficiency, since stealing rate is low and hardly influences performance.

1.6.1 Dealing with Memory Reordering

The presentation of the SALSA algorithm in Section 1.5 assumes sequential consistency [27] as the memory model. However, most existing systems relax se-

quential consistency to achieve better performance. Specifically, according to x86-TSO [35], memory loads can be reordered with respect to older stores to different locations. As shown by Attiya et al. [5], it is impossible to avoid both read-after-write and atomic-write-after-read in work stealing structures, which requires using a synchronization operation, such as a fence or CAS, to ensure correctness. In SALSA, this reordering can cause an index increment to occur after the ownership validation (lines 90, 91 in Algorithm 5), which violates correctness as it may cause the same task to be taken twice, by both the original consumer and the stealing thread.

The conventional way to ensure a correct execution in such cases is to use memory fences to force a specific memory ordering. For example, adding an `mfence` instruction between lines 90 and 91 guarantees SALSA's correctness. However, memory fences are costly and their use in the common path degrades performance. Therefore, we prefer to employ a synchronization technique that does not add substantial overhead to the frequently used **takeTask()** operation. One example for such a technique is location-based memory fences, recently proposed by Ladan-Mozes et al. [26], which is unfortunately not implemented in current hardware.

In our implementation, we adopt the synchronization technique described by Dice et al. [14], where the slow thread (namely, the stealer) binds directly to the processor on which the fast thread (namely, the consumer) is currently running, preempting it from the processor, and then returns to run on its own processor. Thread displacement serves as a full memory fence, hence, a stealer that invokes the displacement binding right after updating the ownership (before line 119 in Algorithm 5) observes the updated consumer's index. On the other hand, the steal-free fast path is not affected by this change.

1.6.2 Experiment Setup

The implementation of the work-stealing framework used in our evaluation does not include the linearizability mechanism described in 1.5.5. We believe that this mechanism has negligible effect on performance; moreover, in our experiment they would not have been invoked because the pool is never empty. We compare the following task pool implementations:

- **SALSA** – our work-stealing framework with SCPools implemented by SALSA.
- **SALSA+CAS** – our work-stealing framework with SCPools implemented by a simplistic SALSA variation, in which every **consume()** and **steal()** op-

eration tries to take a single task using CAS. In essence, SALSA+CAS removes the effects of SALSA's low-synchronization fast-path and per-chunk stealing. Note that disabling per-chunk stealing in SALSA annuls the idea of chunk ownership, hence, disables its low-synchronization fast-path as well.

- **ConcBag** – an algorithm similar to the lock-free Concurrent Bags algorithm [37]. It is worth noting that the original algorithm was optimized for the scenario where the same process is both a producer and a consumer (in essence producing tasks to itself), which we do not consider in this work; in our system no thread acts as both a producer and a consumer, therefore every consume operation steals a task from some producer. We did not have access to the original code, and therefore reimplemented the algorithm in our framework. Our implementation is faithful to the algorithm in the paper, except in using a simpler and faster underlined linked list algorithm. All engineering decisions were made to maximize performance.

- **WS-MSQ** – our work-stealing framework with SCPools implemented by Michael-Scott non-blocking queue [31]. Both **consume()** and **steal()** operations invoke the **dequeue()** function.

- **WS-LIFO** – our work-stealing framework with SCPool implemented by Michael's LIFO stack [30].

We did not experiment with additional FIFO and LIFO queue implementations, because, as shown in [37], their performance is of the same order of magnitude as the Michael-Scott queue. Similarly, we did not evaluate CAFÉ [7] pools because their performance is similar to that of WS-MSQ [6], or ED-Pools [2], which have been shown to scale poorly in multi-processor architectures [6, 37].

All the pools are implemented in C++ and compiled with `-O2` optimization level. In order to minimize scalability issues related to allocations, we use `jemalloc` allocator, which has been shown to be highly scalable in multi-threaded environments [1]. Chunks of SALSA and SALSA+CAS contain 1000 tasks, and chunks of ConcBag contain 128 tasks, which were the respective optimal values for each algorithm (see Section 1.6.6).

We use a synthetic benchmark where 1) each producer works in a loop of inserting dummy items; 2) each consumer works in a loop of retrieving dummy items. Each data point shown is an average of 5 runs, each with a duration of 20 seconds.

The tests are run on a dedicated shared memory NUMA server with 8 Quad Core AMD 2.3GHz processors and 16GB of memory attached to each processor.

1.6.3 System Throughput

Figure 1.4(a) shows system throughput for workloads with equal number of producers and consumers. SALSA *scales linearly* as the number of threads grows to 32 (the number of physical cores in the system), and it clearly outperforms all other competitors. In the $16/16$ workload, SALSA is $\times 20$ faster than WS-MSQ and WS-LIFO, and more than $\times 3.5$ faster than Concurrent Bags.

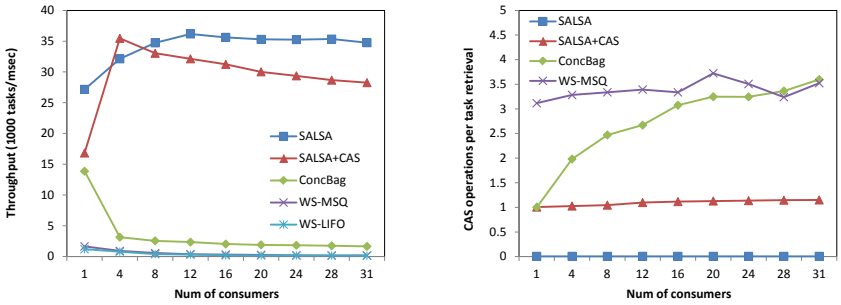
We note that the performance trend of ConcBags in our measurements differs from the results presented by Sundell et al. [37]. While in the original paper, their throughput *drops* by a factor of 3 when the number of threads increases from 4 to 24, in our tests, the performance of ConcBags *increases* with the number of threads. The reasons for the better scalability of our implementation can be related to the use of different memory allocators, hardware architectures, and engineering optimizations.

All systems implemented by our work-stealing framework scale linearly because of the low contention between consumers. Their performance differences are therefore due to the efficiency of the **consume()** operation – for example, SALSA is $\times 1.7$ faster than SALSA+CAS thanks to its fast-path consumption technique.

In contrast, in ConcBags, which is not based on per-consumer pools, every **consume()** operation implies stealing, which causes contention among consumers, leading to sub-linear scalability. The stealing policy of ConcBags algorithm plays an important role. The stealing policy described in the original paper [37] proposes to iterate over the lists using round robin. We found out that the approach in which each stealer initiates stealing attempts from the predefined consumer improves ConcBags' results by 53% in a balanced workload.

Figure 1.4(b) shows system throughput of the algorithms for various ratios of producers and consumers. SALSA outperforms other alternatives in all scenarios, achieving its maximal throughput with equal number of producers and consumers, because neither of them is a system bottleneck.

We next evaluate the behavior of the pools in scenarios with a single producer and multiple consumers. Figure 1.5(a) shows that the performance of both SALSA and SALSA+CAS does not drop as more consumers are added, while the throughput of other algorithms degrades by the factor of 10. The degradation can



(a) System throughput – 1 Producer, N consumers. (b) CAS operations per task retrieval – 1 Producer, N consumers.

Figure 1.5: System behavior in workloads with a single producer and multiple consumers. Both SALSA and SALSA+CAS efficiency balance the load in this scenario. The throughput of other algorithms drops by a factor of 10 due to increased contention among consumers trying to steal tasks from the same pool.

be explained by high contention among stealing consumers, as evident from Figure 1.5(b), which shows the average number of CAS operations per task transfer.

1.6.4 Evaluating SALSA techniques

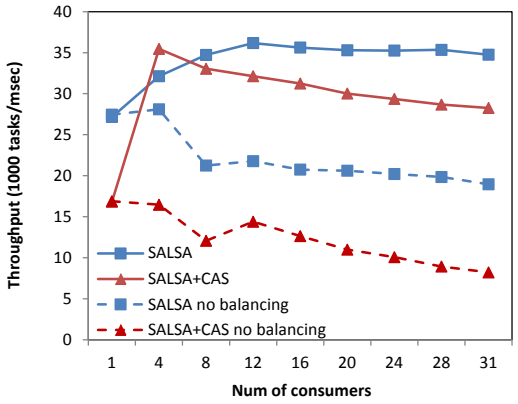


Figure 1.6: System throughput – 1 Producer, N consumers. Producer-based balancing contributes to the robustness of the framework by reducing stealing. With no balancing, chunk-based stealing becomes important.

In this section we study the influence of two of the techniques used in SALSA: 1) chunk-based-stealing with a low-synchronization fast path (Section 1.5.3), and

2) producer-based balancing (Section 1.5.4). To this end, we compare SALSA and SALSA+CAS both with and without producer-based balancing (in the latter a producer always inserts tasks to the same consumer's pool).

Figure 1.6 depicts the behavior of the four alternatives in single producer / multiple consumers workloads. We see that producer-based balancing is instrumental in redistributing the load: neither SALSA nor SALSA+CAS suffers any degradation as the load increases. When producer-based balancing is disabled, stealing becomes prevalent, and hence the stealing granularity becomes more important: SALSA's chunk based stealing clearly outperforms the naïve task-based approach of SALSA+CAS.

1.6.5 Impact of Scheduling and Allocation

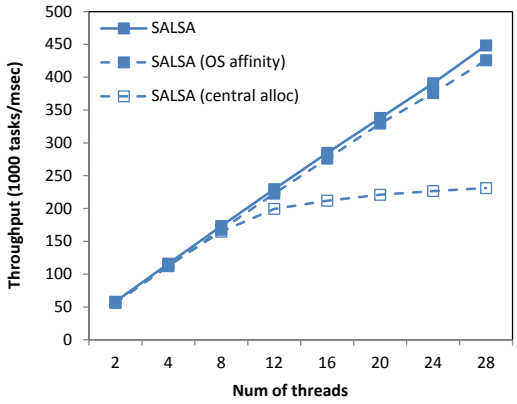


Figure 1.7: Impact of scheduling and allocation (equal number of producers and consumers). Performance decreases once the interconnect becomes saturated.

We now evaluate the impact of scheduling and allocation in our NUMA system. To this end, we compare the following three alternatives: 1) the original SALSA algorithm; 2) SALSA with no affinity enforcement for the threads s.t. producers do not necessarily work with the closest consumers; 3) SALSA with all the memory pools preallocated on a single NUMA node.

Figure 1.7 depicts the behavior of all the variants in the balanced workload. The performance of SALSA with no predefined affinities is almost identical to the performance of the standard SALSA, while the central allocation alternative looses its scalability after 12 threads.

The main reason for performance degradation in NUMA systems is bandwidth

saturation of the interconnect. If all chunks are placed on a single node, every remote memory access is transfered via the interconnect of that node, which causes severe performance degradation. In case of random affinities, remote memory accesses are distributed among different memory nodes, hence their rate remains below the maximum available bandwidth of each individual channel, and the program does not reach the scalability limit.

1.6.6 Chunk size influence

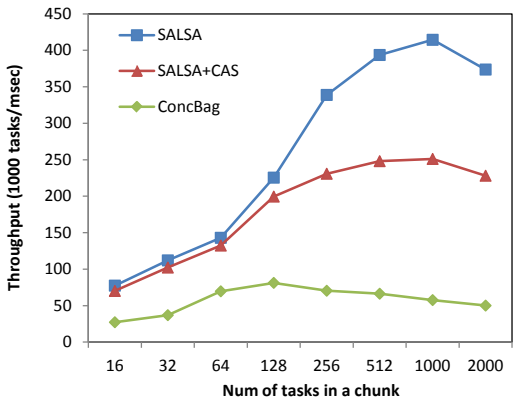


Figure 1.8: System throughput as a function of the chunk size.

Figure 1.8 shows the influence of chunk size on system throughput for the chunk-based algorithms SALSA, SALSA+CAS and ConcBags in a $16/16$ workload. SALSA variations achieve their best throughput for large chunks with 1000 tasks ($\sim$ 8KB size in 64-bit architectures). The optimal chunk for ConcBags includes 128 tasks. We believe that ConcBags is ineffective with large chunk sizes since its consumers linearly scan a chunk when seeking a task to steal. In contrast, SALSA keeps the index of the latest consumed task in the chunk node, and therefore its consume operations terminate in $O(1)$ steps for every chunk size. In our evaluation in section 1.6 we used optimal chunk sizes for each algorithm.

1.7 SALSA correctness

1.7.1 Definitions

First we define constants and definitions that are used in the section.

- A — The system as describe in Section 1.4, when using SALSA pools as the SCPool.
- n — The number of consumers in A.

Definition 1. (Referring Node) *A node is the* referring node *of chunk C if that node points to C, and is in a chunk list of C's owner.*

We now define what we shall call the *commit points* of A, we will later show that these points are the linearization points of A.

Definition 2. *The commit points of A are as follows:*

1. *For a* **put***() operation, the* commit point *is the assignment in line 58 of the* **put***() function.*
2. *For a* **get***() operation that returns a task, the* commit point *is the point where the idx of the* referring node *is increased to include the returned task. More specifically:*
   - *If the task T is returned by* **consume***(), the* commit point *is line 90 of the* **consume***() if the chunk containing T is owned by the consumer executing this* **consume***() operation, and otherwise, it is line 131 executed by a stealing consumer before it removes the chunk from the current consumer's node in line 132*
   - *If the task is returned by* **steal***() and the new node added to the list in line 131 has a higher idx than the node it replaces, then the* commit point *is line 131.*
   - *If the task is returned by* **steal***() and the new node added to the list has the same idx as the node it replaces in line 131, it means that the idx of the replaced node has been incremented between lines 119 and 131. In this case the* commit point *is at the time the idx was increased to its current value. This may be either in line 90 or line 131, depending on the operation (consume() or steal()) executed by the consumer that increments it.*

Definition 3. (taken) *A task T is* taken *at a given time if the idx of the* referring node *of the chunk containing T is greater than or equal to the slot of this task.*

Note that if a task T is returned, then the commit point of the **get**() operation that returns T is the point where the task is *taken*.

Definition 4. (empty) *A task pool is* empty *at a given time t, if all tasks that were added to the pool by* **put***() operations that passed their commit point before time t are* taken *at time t.*

Definition 5. *Let c be the consumer owning a SALSA SCPool, then c's SCPool is* non-empty *if there is a chunk owned by c that contains tasks which are not* taken*.*

1.7.2 Lock-freedom

For the purpose of the proof, we refer to the first part of the **steal**() operation (lines 109 to 118) as *part I* of the operation and to the second part (lines 119 to 138) as *part II* of the operation.

From Definition 5, Definition 4 and the fact the each chunk is owned by a consumer we can reach the following observation:

Observation 1. *If the task pool is not* empty*, then at least one SALSA SCPool is* non-empty*.*

Lemma 1. *If a chunk owned by a consumer c contains a task, then that chunk is accessible from one of the lists in c's SCPool.*

Proof. If c is the first owner of this chunk than that chunk was inserted to c's pool by a producer in line 71. Otherwise, c stole this chunk, and before the changing ownership in line 116, c pointed to this chunk in line 115 and later replaced to node pointing to that chunk in line 131. Therefore, this chunk is accessible via c's SCPool during the time c is the chunk's owner. □

Lemma 2. *If a consumer successfully finishes* part I *of the* **steal***() operation (i.e., succeeds in the CAS in line 116) and later finishes the operation, then in the duration of this* **steal***() operation, a task becomes* taken*.*

Proof. First we note that before the consumer finishes *part I*, it first checks that there is a task in the current chunk, and stores the index of that task in $prevIdx$ (line 113). If the idx as read in line 119 is bigger than $prevIdx$, a task was taken in

the duration of this operation and we are done. Otherwise, idx as read in line 119 is equal to $prevIdx$ and therefore the consumer will reach line 128. In this case the new node replacing the old node will have idx greater than $prevIdx$, and so the task in $prevIdx$ is taken and we are done. □

Lemma 3. *If a consumer fails to finish* part I *of the* **steal***() operation (i.e., fails the CAS in line 116)* n *times on a SCPool that is non-empty when the operation begins, then there is another consumer that takes a task from the task pool during the time interval spanning those* n *failed attempts.*

Proof. Since we assume the SCPool is not empty when the operation begins, then by Lemma 1 there is a list containing a non-empty chunk owned by the victim in the victim's SCPool. Therefore if no chunk is found in line 109 then either a concurrent **consume()** operation took a task in which case we are done, or another **steal()** operation successfully stole a chunk from this SCPool.

Otherwise a chunk is found and the consumer may fail to finish *part I* of the **steal**() operation on a non-empty SCPool in the following cases:

1. The **if** statement in line 113 is true because the chunk does not contain a task. However since, a chunk containing task was chosen in line 109, at least one task was taken from this chunk after it was chosen, and we are done.

2. The **if** statement in line 111 is true. In this case, a stealable chunk was found, but another **steal()** operation successfully stole the chunk before the chunk was read.

3. The **if** statement in line 116 is true. In this case, the **steal**() operation fails because another consumer stole this chunk.

If a task was taken in the period spanning the n operations, we are done. Otherwise, there are n operations by other consumers that successfully stole a chunk, i.e., there are n operations that finished *part I*. Since there are only $n-1$ consumers other than the consumer that failed, we conclude that there is at least one consumer that completed *part II*. Therefore, by Lemma 2, some task was taken during this time. □

Lemma 4. *If a consumer returns* $\perp$ *in* n **steal***() operations on a non-empty SALSA SCPool, then there is a consumer that takes a task from the task pool during that time interval.*

Proof. If the consumer returns $\perp$ because it fails to finish *part I* n times, then by Lemma 3, a task was taken during that time period. Otherwise, at least one of its n steal operations successfully finishes *part I* of the **steal**() and returns $\perp$ in *part II*. By Lemma 2, a task was taken by some consumer during this time interval. □

Lemma 5. *If a consumer returns $\perp$ in n* **consume***() operations on a non-empty SALSA SCPool, then there is a consumer that takes a task from the task pool during that time interval.*

Proof. A **consume**() operation may return $\perp$ in two cases:

1. No chunk with a task was found and $\perp$ was returned in line 82. In this case, no task was found in the SCPool, but since we assume that this SCPool was non-empty when the operation started, we know that the chunk containing this task was stolen by some other consumer.

2. If a chunk with a task was found, and **takeTask**() returned $\perp$. This may happen only if another consumer stole the chunk.

In both cases there was some other consumer that stole a chunk. If this occurs n times, then we know that there are n operations that finished *part I*. Since there are only $n - 1$ consumers other than this consumer, we conclude that there is at least one consumer that finishes *part II*, i.e. returns from its **steal**() operation. Therefore, by Lemma 2, there is a consumer that takes a task. □

Lemma 6. *If* **checkEmpty()** *returns false because the if in line 35 is true $2n$ times, then there is a consumer that takes a task during that time interval.*

Proof. If **checkEmpty()** returns false because of the if in line 35, then some consumer has cleared *emptyIndicator* during the execution of **checkEmpty()**. This can happen only when a consumer successfully steals a chunk or takes a task from a chunk. By Lemma 4, if the first case occurs more than $n - 1$ times, a task is taken and we are done. Otherwise, there are at least n operations that take a task and clear *emptyIndicator*. At most $n - 1$ of these operations were invoked before **checkEmpty()** began. Therefore, at least one of the n operations that take tasks began after the **checkEmpty()** operation began and cleared *emptyIndicator* before it ended. Since this operation takes the task before it cleares *emptyIndicator*, it takes the task before **checkEmpty()** ends, and the lemma follows. □

Claim 1. *If a* **get()** *operation runs for* $5n$ *iterations in* A*, then a task is taken by some consumer in the system during those iterations.*

Proof. The **get()** operation is a loop. In every iteration of the while loop in lines 13-21 it calls **consume()** on the local SCPool, then **steal()** $n-1$ on the other pools, and finally **checkEmpty()**. When **consume()** or **steal()** return a task, this task is returned by the **get()** operation. If **checkEmpty()** returns true, then the **get()** operation returns $\perp$.

Consider a **get()** operation that does not return after $5n$ loop iterations. At the end of each iteration, **checkEmpty()** returns false. If it returns false $2n$ times because of the if in line 35, then by Lemma 6 a task is taken and we are done. Otherwise, the are at least $3n$ iterations in which the task pool contained a task when **checkEmpty**() was called. In each of those iterations, there are three cases: (1) the consumer found the task pool non-empty during a its corresponding **steal**() or **consume**(), (2) the task was taken from this task pool by another consumer, (3) the chunk that included that task was stolen. If case (2) happens we are done. Therefore, assume that all $3n$ iterations fall in cases (1) or (3). If (3) happens n times, then at least one of the consumers finishes the **steal**() operation, and by Lemma 2, a task was taken and we are done. Otherwise, then there are at least $2n$ iterations where the task pool is not empty, and therefore by Observation 1 in those iterations there is at least one non-empty SCPool. Thus, in every iteration the consumer performs **consume**() or **steal**() on a non-empty SALSA SCPool, and since at least n of those operations are of the same type, then by Lemmas 5 and 4 a task will be taken by this consumer or by another consumer during that time. □

We now show the if $(n+1)^2$ tasks are *taken* from the pool a task is returned during that time. Note that while it is possible to show a tighter bound on the number of *taken* tasks, we chose to use a higher value for proof clarity.

Lemma 7. *If* $(n+1)^2$ *tasks are* taken *from the task pool in a certain time interval, then in the duration of this interval a task is returned by some consumer.*

Proof. First we show that if $n+1$ tasks are *taken*, then at least one slot is changed to TAKEN during that time. By Definition 3, a task is *taken* after the idx pointing to the chunk containing that task is increased to include this task This may occur in line 90 or 131. After either of these lines is executed, the consumer always reaches a line that changes the slot to TAKEN if it wasn't already changed (lines 92 and 95 in takeTask() and line 134 in steal()). The slot is not changed to TAKEN before the

task is *taken*, since it is only changed after incrementing of idx. Therefore, after a task is *taken* when the consumer incrementing the idx pointing to this chunk finishes its takeTask() or steal() operation, the slot of this task is changed to TAKEN. Since there are n consumers in the system, if $n+1$ tasks are taken, then at least one consumer finished takeTask() or steal() after executing line 92, 95, or 134, and therefore during the time when $n+1$ tasks are *taken* a task is changed to TAKEN.

Therefore when $(n+1)^2$ tasks are *taken* from the pool, we know that $n+1$ slots are changed to TAKEN. We now note that when a slot is changed to TAKEN by a consumer, that consumer returns that task when it completes its **get**() operation. Since we know that $n+1$ slots were changed to TAKEN, and since there are only n consumers in the system, we know that at least one consumer finished its **get**() operation after changing a slot to taken, and therefore returns that task. □

Theorem 1. *A is lock-free.*

Proof. According to Claim 1, if a **get**() operation runs for $5n$ iterations without taking a task, then a task is *taken* by some consumer in the system. By Lemma 7 if $(n+1)^2$ tasks are *taken* a tasked is returned. Therefore after $(n+1)^2 \times 5n$ iterations of **get**() a is be returned. Therefore, the **get**() operation is lock-free. The **put**() operation is trivially wait-free. □

1.7.3 Linearizability

Lemma 8. *Let C be a task chunk and idx_{t_1}, idx_{t_2} be the idx of the* referring node *of C at times t_1, t_2 respectively, s.t. $t_1 < t_2$. Then $idx_{t_1} \leq idx_{t_2}$*

Proof. First we note that an idx field of a node may only increase after it is created (line 90). It therefore remains to consider the case that the new *referring node* pointing to C replaces an old *referring node*. When the *referring node* pointing to C is replaced by a new *referring node* (line 131) the node is created with the previous node's idx or with its $idx+1$ if the $idx+1$'th slot in C is not $\perp$. However, the previous node's owner may increase its idx after it is read by other consumers. Note that this may occur only if this chunk did not contain $\perp$ in the idx'th slot before the chunk changed ownership, since the consumer checks that the next slot in the chunk is not $\perp$ and that it is the owner before incrementing idx (lines 87 and 88 in takeTask() and lines 124 and 125 in steal()). Therefore, we get that if the previous owner may have increased its idx, then a consumer stealing the chunk will create a new node with $idx+1$. And since after a chunk is stolen the previous

owner may increase the idx at most once before it notices that it was stolen and leaves this chunk, the lemma follows. □

We will now prove that A is linearizable. First we show that the *commit points* defined in Section 1.7.1 are well-defined and therefore can be used as the linearization points of A.

Claim 2. *There is exactly one* commit points *in the duration of any* **put***() operation or* **get***() operation that returns a task.*

Proof. For a **put**() operation, it is easy to see that the function always reaches line 58.

For a **get**() that returns a task, the following cases are possible:

- If the task is taken by **consume**(), then line 90 is always executed before the task is returned. However, this line may be executed after the chunk is stolen. In this case a concurrent **steal**() operation might have removed the chunk from the consumer's list (line 132) and before that, pointed to the chunk with a new node that has higher idx (line 131). If this is the case, then the *commit point* is the time of the node replacement in line 131. Note that the other consumer executed this line during the execution of the **consume**() operation - before line 90 and after the chunk is selected in line 86
- If the task taken by **steal**(), by Lemma 8 there are two options:
  - The new node added to the list in line 131 has a higher idx than the node it replaces. In this case, it is obvious that line 131 is executed before the task is returned.
  - The new node added to the list has the same idx as the node it replaces. This may occur only if the idx of the original node is increased after the stealing consumer reads its value in line 119 and before the stealing consumer replaces the node in line 131. Therefore the incrementation of idx is performed in the course of the stealer's **steal**() call.

□

We will show that *commit points* as described above are valid linearization points for **put()** operations, and for **get()** operations that return a task. For **get**()

operations that returns $\perp$, we will show that such a linearization point exist without explicitly specifying it.

The following observation follows immediately from the code in Algorithm 6.

Observation 2. *If a consumer operation that takes the last task in a pool returns a task, this operation clears the emptyIndicator of this pool after taking the task and before starting a new operation.*

Claim 3. *If* **checkEmpty()** *returns true then there is a time between its invocation and its response when the task pool is empty.*

Proof. In every iteration of the loop in line 32 of **checkEmpty()**, the consumer checks that its bit in *emptyIndicator* is set (line 35). If **checkEmpty()** returns true then the *emptyIndicator* was not reset by any consumer after it was set in the first iteration. Note that an operation may take the last task in the pool and then stall before clearing *emptyIndicator*. Since there are $n-1$ consumers other than the consumer running checkEmpty(), there may be up to $n-1$ such operations. Since only $n-1$ consumers may take the last task from a pool without clearing the *emptyIndicator* of that pool (by Observation 2), we can conclude that there is at least one iteration during which no pool changes from non-empty to empty. During this iteration, **checkEmpty()** does not find a task in line 106. Therefore, when that iteration began, the pool was empty and the claim follows. □

Lemma 9. *Let σ be a run and t a time in σ such that all the pending operations that started before time t complete in σ and, assume a consumer c increments the idx field of a node at time t. Then the task pointed by this idx will be returned either by that consumer or another consumer running a concurrent* **get***() operation that started before c incremented the idx field.*

Proof. First we note that operations that start after idx is incremented do not take the task pointed by that idx, since they read the up-to-date idx, which by Lemma 8 never decreases. Therefore, if an operation takes the task pointed by idx after it is incremented, it must be an operation that started before c's operation.

The idx field can be incremented in the takeTask() or steal() functions.

If the idx was incremented in the taskTask() function in line 90 then there are three possible cases:

1. If c is still the owner of the chunk when it reaches line 91, then c will return this task in line 94.

2. Otherwise, if the chunk is stolen before c executes line 91, then c tries to CAS the slot from the task to TAKEN in line 95, and if the CAS is successful, c returns it in line 98.

3. Otherwise, some other consumer c' succeeds in changing the slot to TAKEN in line 134, and returns this task in line 138.

If the idx field is incremented in steal() in line 131 by replacing the old node with a node with higher idx, then c created this node with a higher idx and therefore must have executed line 128, which means that the if in line 124 was true, and the slot did not contain $\perp$. Therefore, c will reach line 134, and will try to CAS the slot from the task to TAKEN. If it is successful it returns the task, and otherwise, some other consumer succeeds, and that consumer returns the task.

□

Claim 4. *Let σ be a run and t a time in σ such that all the pending operations that started before time t complete in σ and the system is* empty *at time t. Then every task that was added to the pool by a put() operation that passed its* commit point *before time t is returned by some get() operation whose* commit point *is before time t.*

Proof. If the system is empty, then by Definitions 4 and 3 the idx of every node is greater than or equal to the the index of the last task in that chunk. By Lemma 9, if the idx is increased then the task in that idx is either returned, or is about to be returned by an active **get**() operation. By the definition of the *commit point* of **get**(), that operation has passed its commit point, since the idx of the node was increased. The claim follows. □

Claim 5. *If a consumer c returns a task T, then there is a* **put***(T) operation that passes its* commit point *before c's* **get***() operation passes its* commit point.

Proof. Before a consumer returns a task, the idx field of the node pointing to the task is incremented. Since both takeTask() and steal() verify that the task is not $\perp$ before incrementing idx, we know that the **put**() operation already passed its commit point before the idx is incremented, and by the definition of the **get**() commit points, the claim follows. □

Lemma 10. *If a consumer c_s steals a chunk from a consumer c_v and this chunk's referring node's idx value is i when c_s reads it in line 119. Then (1) c_v does not*

take tasks from indexes greater than $i+1$ in this chunk unless c_v resteals the chunk. And (2) If c_v attempts to take a task from slot $i+1$ in this chunk it does so using CAS.

Proof. First we note that in **consume**(), after a consumer increments the value of a node's idx, it then checks that it is still the owner of the chunk pointed by that node. If the consumer notices that it is not the case it leaves it (line 97). Therefore, after a successful **steal**(), the previous consumer of the chunk can increase the idx field by at most one. In **consume**(), the consumer takes tasks from the idx'th slot of the chunk (lines 92 and 95), and therefore it does not take tasks from slots larger than $i+1$.

Since c_s reads the idx of c_v's node after changing ownership (line 119), if c_v increases the idx in line 90 after c_s steals the node, c_v notices the ownership change (line 91) and therefore attempts to take the task using a CAS operation in line 95.

If c_v is executing **steal**(), c_v either takes the $i+1$'st task if idx is read by c_s before c_v increments it in line 131 or it takes the i'th task if c_s reads it after it is increased. In both cases, c_v used CAS to take the task. Moreover, since the ownership has changed, c_v does not try to take tasks from this chunk before re-stealing it, since this chunk is not chosen by this consumer if it is not the owner (line 79). □

Lemma 11. *Let c_s be a consumer stealing a chunk from consumer c_v, and let the idx value of the referring node of that chunk be i when c_s reads it in line 119. Then (1) c_s only takes tasks from indexes greater than i; and (2) if another consumer tries to take a task from index $i+1$, then c_s attempts to take that task using CAS.*

Proof. The first task c_s attempts to take is the task at index $i+1$ (line 134). This is done by a CAS operation if there is a task in that slot when c_s reads the contents of the slot in line 123. If the slot is $\perp$ when c_s reads it, c_s may later take this task without a CAS operation if the chunk is not stolen. In the later case, other consumers do not try to take this task unless they steal the chunk, since they may only see this task after c_s changes the ownership, and since after reading a slot, ownership is checked (lines 88 and 125). An exception is in line 125 where the task might be taken in case the ownership changed. However, this is done only if the task was there before the ownership change and therefore c_s is guaranteed to also notice this task. If the chunk is stolen from c_s, then by Lemma 10 c_s takes the $i+1$'st task using CAS.

After c_s takes the first task, it increments idx in line 131 or line 90, and since subsequent **consume**() operations will take tasks from slots $i + 1$ and higher, the lemma follows. □

Lemma 12. *A task in A may be only returned once.*

Proof. Consider a consumer c_s that takes a task. If c_s stole the chunk from another consumer c_v, then by Lemma 10 and Lemma 11, c_s and c_v do not take tasks from the same slot, and if they do, they use CAS. Since only one consumer may succeed in a CAS operation we conclude that a task will be returned by at most one consumer, and since a consumer will not attempt to take the same task twice, as it always takes tasks from $idx + 1$ and always increases idx, a task can be returned only once.

□

Theorem 2. *A is linearizable.*

Proof. We will now show that it is possible to choose the linearization points to be the commit point as defined above. We only show correctness for complete histories. However, since our algorithm is lock-free it is possible to complete pending operations of partial histories so they will be complete. Therefore our proof also holds for partial histories.

From Claim 5 we know that the linearization point of a consumer executing $get()$ that returns T always follows the linearization point of $put(T)$. From Claim 12 we know that for each $put(T)$ operation, at most one $get()$ returns T. From Claim 3 we know that if a $get()$ operation returns $\perp$, then there is a point during its execution in which the pool is empty. From Claim 4 we know that each $put(T)$ operation that preceded a point in which the pool was empty there is a a $get()$, which stats after he linearization point of $put(T)$ that returns T. □

1.8 Conclusions

We presented a highly-scalable task pool framework, built upon our novel SALSA single-consumer pools and work stealing. Our work has employed a number of novel techniques for improving performance: 1) lightweight and synchronization-free produce and consume operations in the common case; 2) NUMA-aware memory management, which keeps most data accesses inside NUMA nodes; 3) a chunk-based stealing approach that decreases the stealing cost and suits NUMA migration

schemes; and 4) elegant producer-based balancing for decreasing the likelihood of stealing.

We have shown that our solution scales linearly with the number of threads. It outperforms other work-stealing techniques by a factor of 20, and state-of-the art non-FIFO pools by a factor of 3.5. We have further shown that it is highly robust to imbalances and unexpected thread stalls.

We believe that our general approach of partitioning data structures among threads, along with chunk-based migration and an efficient synchronization-free fast-path, can be of benefit in building additional scalable high-performance services in the future.

Chapter 2

On Locality Effects in STM

2.1 Introduction

Locality has always been an important aspect of many research fields, such as concurrent data structures, concurrent algorithms, and memory allocators. Indeed, many papers discuss locality and cache-awareness (see Section 2.2). However, somewhat surprisingly, in the field of software transactional memory (STM) [36], not many papers address this subject, and the few that do, only refer to it shortly.

The benefits of locality are well known and consist of two aspects: the first is spacial locality, which affects both single-threaded and multi-threaded applications. The second is cache-contention, which affects multi-threaded programs. In Section 2.4.3 we examine these two locality effects, and run micro-benchmarks in order to understand them and their impact on performance better. Our micro-benchmarks show that both effects can have a potentially large impact on performance.

In STM systems, one way to achieve locality is by storing the meta-data inline with the data. This approach was suggested in the past in McRT-STM [34] for word-based systems, and in [18] for object-based systems, but the locality effects were not evaluated or explained in detail. In this chapter we use an approach similar to the one used in McRT-STM and evaluate the effects of this approach. We create a version of TL2 that emulates this approach by storing locks inline with the data, instead of storing them in a global lock-table as done in most STMs.

Storing meta-data inline with the data has several advantages:

- Avoiding global meta-data can reduce so called “false” cache contention

caused due to multiple unrelated meta-data items sharing the same cache line.

- Because the data is stored next to the meta-data, fetching the meta-data may fetch the data to the cache and thus reduce the number of cache misses.
- Whereas storing meta-data inline with the data uses the minimal amount of memory necessary, storing meta-data in a global hash table has a large memory overhead.
- Storing meta-data in a global table may result in false conflicts if one location is used for more than one lock. This is not a problem with local meta-data, since there is exactly one lock per allocation.

We look at TL2 [16] as a test case and we run benchmarks from the STAMP benchmark suite in order to evaluate the effect of storing meta-data locally on an STM system. Our results show a 20-230% speedup when comparing local meta-data to a global lock-table. We also see a decline in the number of cache misses that may explain this difference. In summary, our contributions in this chapter are:

- Understanding the two locality effects.
- Evaluating the effects of meta-data locality on an STM system.
- Understanding how meta-data locality may help improve STM systems.

2.2 Related Work

The effect of data locality on performance is the subject of many works. In [13], the authors recognize data locality as an important problem. The authors suggest that data should be clustered in memory in a data-structure aware manner. They suggest a new version of *malloc* which can allocate related data in the same cache block, thus reducing the number of accesses to memory. Other works such as [38], [29] suggest improving locality by compiler based optimizations, while works such as [33] and [12] use programmer "hints" to improve data locality of both sequential and parallel programs.

Many of the STM systems for unmanaged environments are lock-based ([16], [19], [17], [34]). Locks are used to protect shared memory locations in order to

make sure that the transactions do not conflict before a transaction is allowed to commit (see section 2.4.1). Most of these algorithms have been implemented by a global lock-table ([16], [19], [17]). In our work we focus on TL2 [16], a lock-based STM system, in which locks are only acquired at commit time, but are validated before and after every read, thus making the overhead of accessing locks significant.

In this chapter we evaluate the idea of storing locks inline instead of using a global lock table. This idea of inline locks in STM systems was first presented in McRT-STM [34]. This STM system had an optional in-line locks implementation, which allocated the locks of small objects in-line with the data. This is done by using a custom memory allocator. The effect of in-line locking vs. global lock table were tested by three micro benchmarks and showed improvement in two of the benchmarks. However, the reason for those improvements were not further looked at, and the effect was not tested using more realistic benchmarks.

Concurrently and independently to our work, Mannarswamy and Govindarajan [28] suggested compiler transformations for reducing cache-misses in STMs. They showed that cache-misses of the STM system account for a large number of the total cache-misses of the STAMP benchmarks. They showed that in most benchmarks most of the cache-misses originate from accessing locks. The solution they gave for this problem is holding the lock in-line with the data, similarly to McRT-STM.

2.3 Locality in different architectures

In this section we discuss the different locality effects, and run micro-benchmarks in order to see those effects.

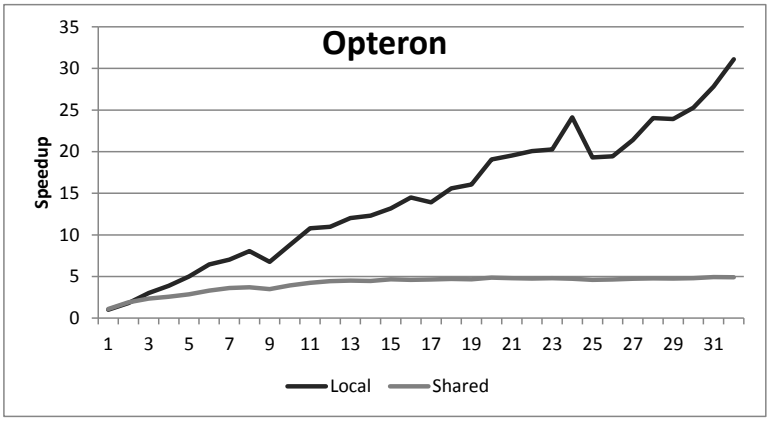
We run the micro-benchmarks on two systems:

1. An AMD Opteron system with eight 4-core CPUs with NUMA layout and 128GB of RAM.

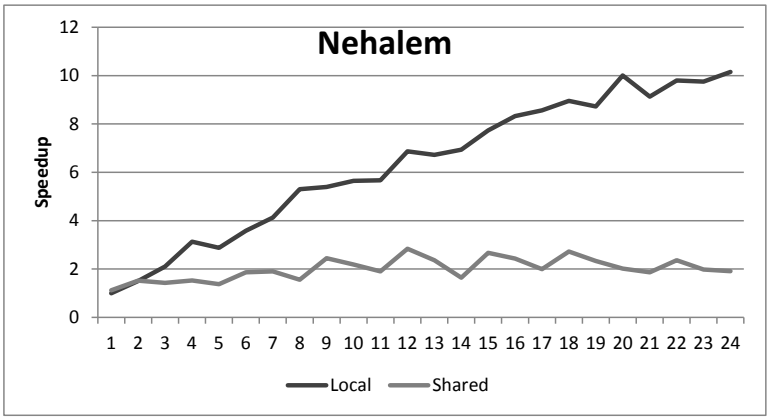
2. An Intel Nehalem system with two 6-core hyperthreaded CPUs (a total of 24 hardware threads) and 80GB of RAM.

In the first benchmark we check how contention affects performance and in the second we check how spacial data locality affects performance.

2.3.1 Memory contention effect on different architectures



(a) Opteron



(b) Nehalem

Figure 2.1: Contention effect - speedup.

The purpose of this benchmark is to show the effect of sharing data among threads. In this simple benchmark all threads access an integer in a 80%/20% R/W ratio. We run the benchmark in two modes:

- Fully local - all threads access different integers, so no data is shared.
- Fully shared - all threads access the same integer, so that all read and write operations are to shared data.

	Opteron - 32 threads		Nehalem - 24 threads	
	Fully Shared	Fully Local	Fully Shared	Fully Local
L1 miss rate	0.987	0.005	7.826	0.446
Total miss rate	1.129	0.001	6.116	0.240

Table 2.1: Contention effect - cache miss rates (%).

We run the benchmark on both systems with a different number of threads, up to the hardware thread limit of the machine. In each test we run a constant number of operations which we divide among the threads. Thus, if the threads do no interrupt each other, we expect the speedup to be linear in the number of threads. In Figure 2.1 we show the result of this benchmark. The x-axis is the number of threads and the curve shows the speedup of the benchmark relative to the run time of the benchmark with one thread.

We can see that there is a major difference between the runs. This can be explained by the difference between the latency of accessing a value in the L1 cache and a value located in the memory. In Table 2.1 we can see the L1 data cache loads miss rate and the overall miss rate (#of LLC load misses/#L1 loads), where LLC is the last level cache. It is clear that there is a major difference between the two modes, and this verifies our assumption.

2.3.2 Spacial locality effect

In this benchmark we test the effect of spacial locality on performance. Again we implemented a simple benchmark with two modes. In the first mode, which we call no-spacial-locality, each thread performs reads from different memory locations, where each read accesses two cache lines that are far enough apart so that the data is not fetched together from memory to cache. The threads do not share their data in order to disable contention effects.

The second mode, spacial-locality, is identical to the first, except that every even operation is performed on the same cache as like the preceding operation. In this case, half of the operations access data that resides in the cache.

The results of this benchmark on 16 threads can be seen in Figure 2.2. Here we can see that both machines show a 70-99% runtime speedup. In Table 2.3.2 we can see that, as expected, the no-spacial-locality mode has about twice as much misses as the spacial-locality mode.

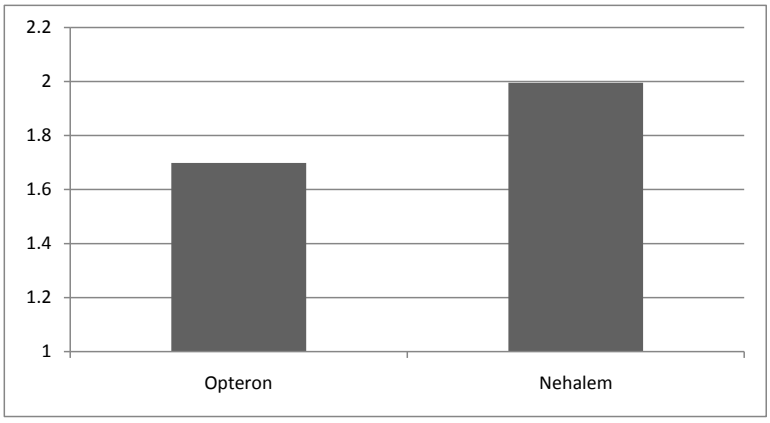


Figure 2.2: Prefetch effect - speedup.

	Opteron		Nehalem	
Spacial Locality	No	Yes	No	Yes
L1 miss rate	29.57	13.09	56.72	23.58
Total miss rate	29.88	13.24	0.70	0.47

Table 2.2: Prefetch effect - cache miss rate (%).

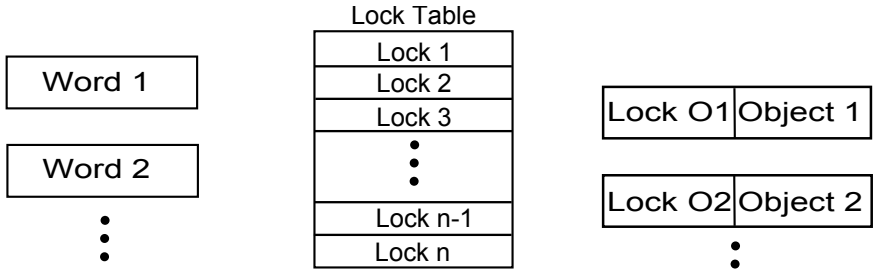
2.4 Locality in STM

In this section we present an alternative method to store meta-data in a local-aware manner and discuss the effects it may have on the performance of the STM system. We look at TL2 [16] as a case study, and implement a variant of it that stores locks near the data instead of storing them in a global hash-table.

2.4.1 Background

In many lock-based STM implementations, like TL2[16], locks are held in a global lock-table which is usually implemented as an array-based hash table. Each word or group of several words in the memory is mapped to an entry in the array that contains the lock for that word by a simple hash function (see Figure 2.3(a)). In case of collisions, it is possible for two words to share a lock, thus causing transactions to abort due to false-conflicts.

In TL2, the lock is checked before and after each read, if a read is successful, then the address read is added to the transaction's *read-set*. When writing to an



(a) Lock table (b) Lock with data

Figure 2.3: Storing locks in a table vs. storing locks with the data.

object, the object is buffered in a set called the transaction's *write-set*. When the transaction commits, locks are acquired for all the addresses in the write-set, then, the lock is validated again for each entry in the read set, and finally the data in the write-set is written back to memory and the locks are released.

2.4.2 Local Meta-data implementation

While many previous STM systems store meta-data in a global table (see Figure 2.3(a)), we store meta-data, such as locks, near the data itself, as depicted in Figure 2.3(b). A similar approach was used in McRT-STM [34]. Since our goal in this work is to see the effects of locality on STM, our implementation is not a full working system, but rather an emulation of such a system. A more detailed description on how such system can be fully implemented is found in [34].

There is a technicality related to changing the location of the lock - it requires changing the memory allocation library, as described in [34]. Specifically, for our experiments, we alter the implementation supplied with the STAMP benchmark suite [11]. In this implementation, every call to *malloc* is wrapped with a function that provides the block size This is done so that when a block is freed it will be possible to free the locks associated with each word of this block. In our implementation, we change the wrapper function to also include the lock, (similarly to [34]). We also adapt the TL2 implementation to work with this change.

In order to locate the lock when reading or writing a value, the STM system must know where the start of the block is. This may be a problem if the operation is done on a field in a struct or a cell in an array, since only the pointer to the data is passed to the STM system. In order to bypass this problem, we changed the implementation of the read and write macros so that they will get a pointer to the

start of the block in addition to the actual data location. This change is for the sake of testing only. As described in [34], it is possible to implement a real STM system that stores the meta-data as we do even without passing the address of the lock by changing the implementation of the memory allocator.

As described in [34], when the allocated objects are large, as in the case of arrays, it is better to use a lock-table instead of a lock per object in order to avoid false-conflicts. We do not discuss this aspect in this chapter, and instead restrict our attention to benchmarks with small objects.

In addition, we also altered the read-set implementation. Originally it was implemented as an array with an initial capacity of 8192. Each read of a memory location inserted the address of the lock to the last free slot in this array, and therefore reading from the same address twice would result in two identical array entries. To eliminate this redundancy, we changed the implementation of the read-set and implemented it as a hash-set with an O(1) add operation, an O(n) reset operation, and an O(n) object iteration operation. This change helps both the local meta-data and the global lock-table implementations. In fact, the global lock-table implementation has a greater benefit from this change than the local meta-data implementation.

2.4.3 Local Meta-data advantages

Storing meta-data locally has several advantages over the global lock-table implementation:

Local meta-data: In the global lock-table implementation, adjacent locks may share the same cache line; this is a case of *false-sharing*. In this case a thread that writes to one lock may cause cache invalidations to other threads that read adjacent locks. In contrast, in the local meta-data implementation, there is no such false-sharing. The local approach therefore reduces the contention on the cache, which we expect to yield a similar effect to the one we saw in Section 2.3.1.

Spacial & temporal locality: As described in Section 2.4.1, in TL2, the lock is checked before and after each read. When a global table is used the lock resides in a global hash-table which is far from the data, and therefore each read requires fetching at least two different memory location to the cache. In the local meta-data implementation, when a lock is read, the data associated with that lock is often also

fetched to the cache. Since the next action after reading a lock is usually reading the data (unless the transaction aborts before the read), we will not get a cache miss here. This is particularly significant if the data set is large, and most accesses are to objects that are not in the already in the cache. Here we expect to see speedup due to the spacial memory locality effect that we saw in section 2.3.2.

Lock granularity: In the local meta-data implementation, the locking granularity is not arbitrary as it is with the fixed-granularity implementation of TL2. Rather, we keep one lock per object allocated by malloc. This may be an advantage in cases such as structs, which are semantically one object, since when we access one field in a struct there is high probability of accessing additional fields in the same struct. Also, in this scenario, accessing several fields of the same struct will add only a single lock to the read-set, where originally there was one lock per field. Therefore, in the local meta-data implementation, the read-set is smaller and therefore consumes less memory and takes less time to validate before commit. The downside of this method is that for very big objects, like arrays, one lock may be too coarse and cause false conflicts. Indeed, as noted above, it is not recommended to use this approach for large objects.

Memory consumption and false conflicts: Unlike a global table, where there inherently must be many empty table entries to reduce the probability of collision, in the local meta-data implementation there is exactly one lock per allocation, therefore, memory consumption is lower. Moreover, with a global table, two objects may be mapped to the same lock and thus cause a *false-conflict* between transactions that access different objects leading to spurious aborts. In our experiments we increased the size of the lock-table from 2^{20} to 2^{25} to minimize such cases. Furthermore we experimented on machines with ample memory, minimizing the effect of the big tables.

2.5 Benchmarks

We evaluate the effect of local meta-data using some of the STAMP[11] benchmarks.

We do not use benchmarks that have large shared objects, since as mentioned

above, local meta-data is not a good solution for such objects - they are best supported by ”falling back” on a global lock table[34]. The benchmarks we run are: vacation, yada and intruder. We use the recommended configurations suggested in [11]. Suffixes of *low* and *high* indicate the relative amount of contention, and the ‘+’ symbol indicates a larger input size.

We run the benchmarks on both systems (see 2.3), each benchmark was run with 1,2,4,8 and 16 threads on both machines, and 32 threads on the AMD machine. Each data point is the average of 5 runs.

The first benchmark is vacation, an implementation of an online transaction system emulating a travel reservation system. Each transaction performs several operations on the database, which is implemented as a set of red-black trees. We found that in most cases, contention in this benchmark is very low and the majority of transactions commit. The second benchmark is intruder, a network intrusion detection system. This system has two main data-structures, a FIFO queue and a balanced tree. This benchmark has short transaction with high contention. The third benchmark is Yada, which implements Ruppert's algorithm for Delaunay mesh refinement. This benchmark has long transactions with medium contention.

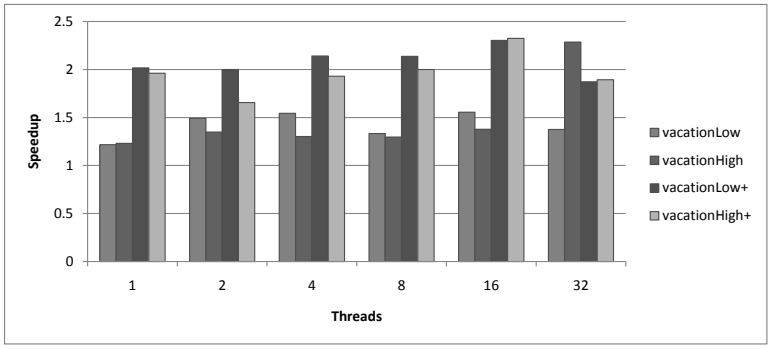
2.5.1 Results

Figures 2.4 and 2.5 show the speedup of the local meta-data implementation over the global lock-table implementation for the all three benchmarks on both machines.

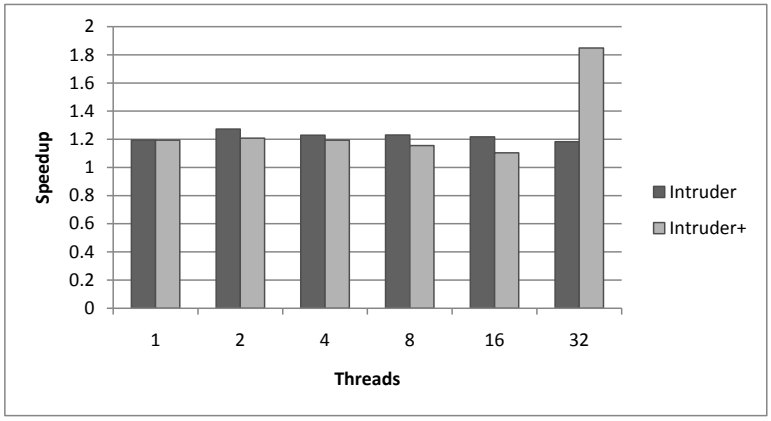
For the vacation benchmarks we can see that while using local meta-data is better for all cases, the speedup is greater when the input is larger, this may be because in those cases the chance of a lock to already be in the cache before the it is accessed is smaller compared to the benchmark with smaller data sets. This leads to a bigger advantage for the local meta-data implementation, since the locality effects are more relevant when the lock is not already in the cache.

We can see that the improvement is not just because of less false-conflicts, since there is a significant improvement even when only one thread is used. Thus we can assume that this speedup is mainly due to locality effects.

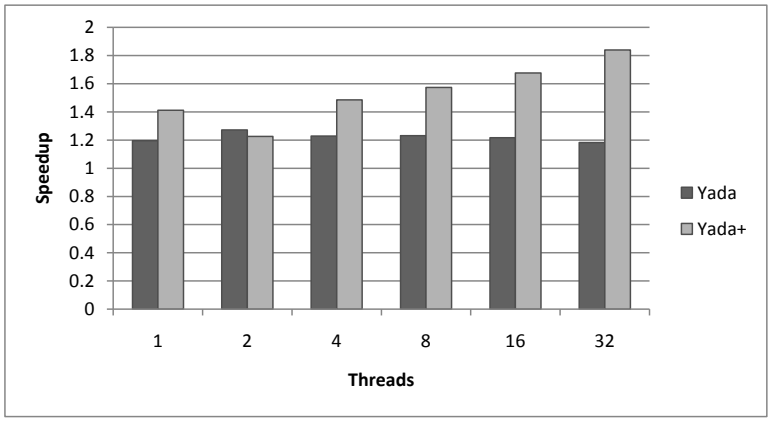
For the intruder benchmark we can see a 20% speedup on the Opteron machine and a 10%–20% speedup in the Nehalem machine. For the yada benchmark we see a 20% speedup for yada on the Opteron machine. For yada+ we see a 40%–80% speedup, the incline in speedup may be caused by a larger number of aborts in the



(a) vacation speedup - Opteron

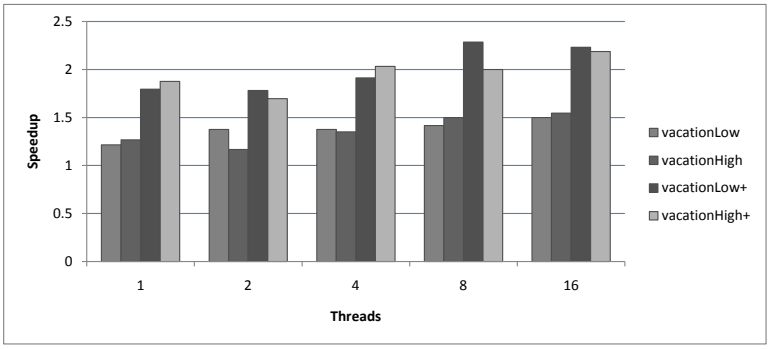


(b) intruder speedup - Opteron

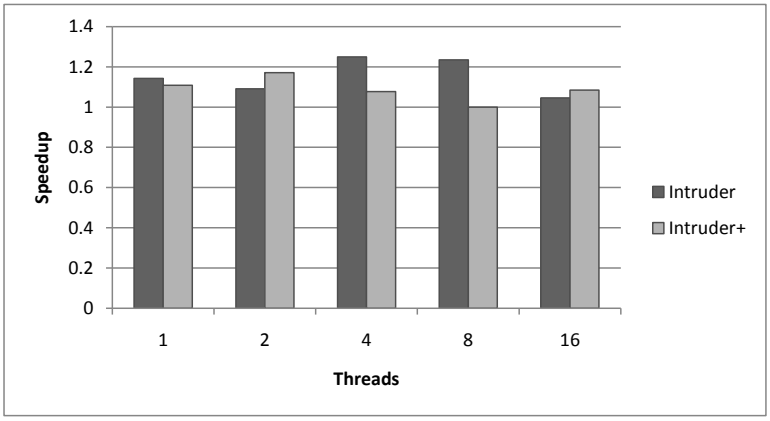


(c) yada speedup - Opteron

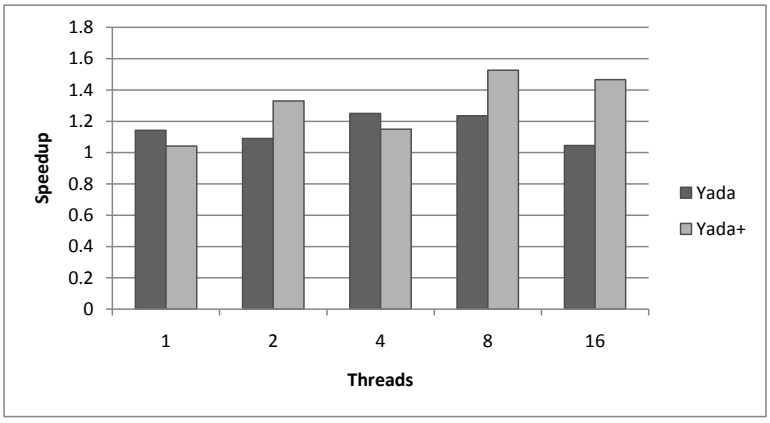
Figure 2.4: STAMP speedup on Opteron.



(a) vacation speedup - Nehalem



(b) intruder speedup - Nehalem



(c) yada speedup - Nehalem

Figure 2.5: STAMP speedup on Nehalem.

normal version due to false-conflicts, the probability for such aborts increase as more threads are run.

2.5.2 Readset

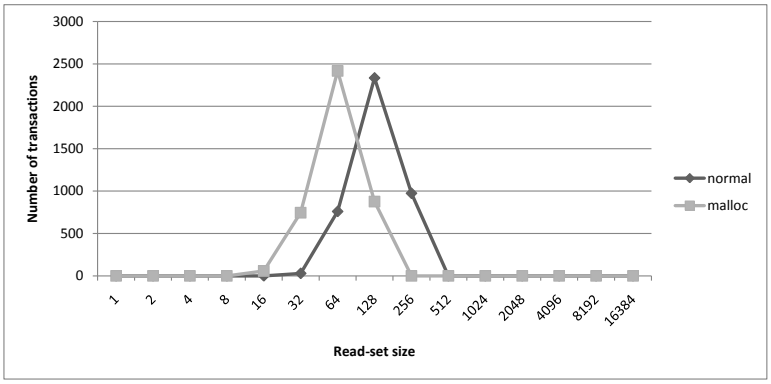


Figure 2.6: VacationHigh read-set size distirbution.

As we stated in Section 2.4.3 the size of the read-set is also affected by our change due to the changed granularity. While we believe the effect of this on the runtime is minor, we show in Figure 2.6 the read-set size for the vacationHigh benchmark with one thread on the Opteron machine. The x-axis is the size of the read-set divided to buckets, the y-axis is the number of transactions that committed with that read-set certain size. It can be seen that our the local meta-data system's read-set is about half the size of the original TL2 implementation.

2.5.3 Cache effects

In this section we will see how locality effects in our the local meta-data STM system reflect in cache misses. To see this we run the STAMP tests we ran in Section 2.5.3. Most runs were with one thread so other effects that may occur due to aborts will be disabled. We also run vacation with 16 threads since the aborts rate there are low. We then read the performance counters for the L1 and last-level-cache loads and load-misses and compare the results between the global lock-table and the local meta-data implementations.

Table 2.3 shows the cache miss rate for three benchmarks with one thread, and for vacationHigh with 16 threads. It can be seen that there are less cache misses

when the metadata is inline. However, this change seems to be small. This can be explained by the fact that in those benchmarks a significant part of the code is outside of the STM system, which makes the STM system less significant. Nevertheless, those changes, while small, can have a noticeable effect on the performance as accessing memory has high latency penalty, and therefore may explain some of the changes in latency in our tests.

(a) vacationHigh 1 thread

	Opteron		Nehalem	
Meta data	Global	Local	Global	Local
L1 miss rate	1.81	1.38	3.87	2.73
Total miss rate	0.81	0.65	1.21	1.06

(b) vacationHigh 16 threads

	Opteron		Nehalem	
Meta data	Global	Local	Global	Local
L1 miss rate	1.84	1.38	3.80	3.23
Total miss rate	0.88	0.72	1.33	1.19

(c) intruder 1 thread

	Opteron		Nehalem	
Meta data	Global	Local	Global	Local
L1 miss rate	0.66	0.37	1.42	0.87
Total miss rate	0.10	0.06	0.34	0.20

(d) yada 1 thread

	Opteron		Nehalem	
Meta data	Global	Local	Global	Local
L1 miss rate	0.99	0.66	2.78	1.69
Total miss rate	0.28	0.18	0.47	0.30

Table 2.3: STAMP - cache misses (%).

Bibliography

[1] www.facebook.com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919.

[2] Yehuda Afek, Guy Korland, Maria Natanzon, and Nir Shavit. Scalable producer-consumer pools based on elimination-diffraction trees. In *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part II*, Euro-Par'10, pages 151–162, 2010.

[3] Yehuda Afek, Guy Korland, and Eitan Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. In *Principles of Distributed Systems*, Lecture Notes in Computer Science, pages 395–410.

[4] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '98, pages 119–129, 1998.

[5] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. pages 487–498, 2011.

[6] Dmitry Basin. Café: Scalable task pools with adjustable fairness and contention. Master's thesis, Technion, 2011.

[7] Dmitry Basin, Rui Fan, Idit Keidar, Ofer Kiselov, and Dmitri Perelman. Café: scalable task pools with adjustable fairness and contention. In *Proceedings of the 25th international conference on Distributed computing*, DISC'11, pages 475–488, 2011.

[8] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. A case for numa-aware contention management on multicore systems. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, USENIXATC'11, 2011.

[9] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46:720–748, September 1999.

[10] Anastasia Braginsky and Erez Petrank. Locality-conscious lock-free linked lists. In *Proceedings of the 12th international conference on Distributed computing and networking*, ICDCN'11, pages 107–118, 2011.

[11] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[12] Rohit Chandra, Anoop Gupta, and John L. Hennessy. Data locality and load balancing in cool. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '93, pages 249–259, New York, NY, USA, 1993. ACM.

[13] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, PLDI '99, pages 1–12, New York, NY, USA, 1999. ACM.

[14] Dave Dice, Hui Huang, and Hui Yang. Asymmetric dekker synchronization. Technical report, Sun Microsystems, 2001.

[15] Dave Dice, Virendra J. Marathe, and Nir Shavit. Flat-combining numa locks. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, 2011.

[16] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *Proceedings of the 20th international conference on Distributed Computing*, DISC'06, pages 194–208, Berlin, Heidelberg, 2006. Springer-Verlag.

[17] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. In *Proceedings of the 2009 ACM SIGPLAN conference*

on Programming language design and implementation, PLDI '09, pages 155–165, New York, NY, USA, 2009. ACM.

[18] Robert Ennals. Cache sensitive software transactional memory. Technical report.

[19] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 237–246, New York, NY, USA, 2008. ACM.

[20] Anders Gidenstam, Håkan Sundell, and Philippas Tsigas. Cache-aware lock-free queues for multiple producers/consumers and weak memory consistency. In *Proceedings of the 14th international conference on Principles of distributed systems*, OPODIS'10, pages 302–317, 2010.

[21] Elad Gidron, Idit Keidar, Dmitri Perelman, and Yonathan Perez. SALSA: scalable and low synchronization NUMA-aware algorithm for producer-consumer pools. In *Proceedinbgs of the 24th ACM symposium on Parallelism in algorithms and architectures*, SPAA '12, 2012.

[22] Danny Hendler, Yossi Lev, Mark Moir, and Nir Shavit. A dynamic-sized nonblocking work stealing deque. Technical report, 2005.

[23] Danny Hendler and Nir Shavit. Non-blocking steal-half work queues. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, PODC '02, pages 280–289, 2002.

[24] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12:463–492, July 1990.

[25] Moshe Hoffman, Ori Shalev, and Nir Shavit. The baskets queue. In *Proceedings of the 11th international conference on Principles of distributed systems*, OPODIS'07, pages 401–414, 2007.

[26] Edya Ladan-Mozes, I-Ting Angelina Lee, and Dmitry Vyukov. Location-based memory fences. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 75–84, 2011.

[27] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, pages 690–691, 1979.

[28] Sandya Mannarswamy and Ramaswamy Govindarajan. Making stms cache friendly with compiler transformations. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 232–242. IEEE, 2011.

[29] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Trans. Program. Lang. Syst.*, 18(4):424–453, 1996.

[30] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15:491–504, June 2004.

[31] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC '96, pages 267–275, 1996.

[32] Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. Using elimination to implement scalable and lock-free fifo queues. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '05, pages 253–262, 2005.

[33] James Philbin, Jan Edler, Otto J. Anshus, Craig C. Douglas, and Kai Li. Thread scheduling for cache locality. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, ASPLOS-VII, pages 60–71, New York, NY, USA, 1996. ACM.

[34] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Ben Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *PPOPP*, pages 187–197, 2006.

[35] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, pages 89–97, 2010.

[36] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[37] Håkan Sundell, Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. A lock-free algorithm for concurrent bags. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 335–344, 2011.

[38] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, PLDI '91, pages 30–44, New York, NY, USA, 1991. ACM.

ניצול לוקליות ומערכות עם גישה לא אחידה לזכרון בספריות סקלביליות מקביליות

אלעד גדרון

ניצול לוקליות ומערכות עם גישה לא אחידה לזכרון בספריות סקלביליות מקביליות

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
מגיסטר למדעים במדעי המחשב

אלעד גדרון

הוגש לסנט הטכניון – מכון טכנולוגי לישראל

אלול ה'תשע"ב חיפה ספטמבר 2012

המחקר נעשה בהנחיית פרופ' עדית קידר מהפקולטה להנדסת חשמל בפקולטה למדעי המחשב.

אני מבקש להביע את תודתי למנחה שלי, פרופ. עדית קידר, שליוותה אותי במהלך עבודתי ולימדה אותי איך עושים מחקר. כמו כן אני רוצה להודות לדמיטרי פרלמן, שהיה חבר ושותף למחקר, על העבודה המשותפת והעזרה הגדולה במהלך המחקר . לבסוף, אני מודה לבת-זוגתי רותי שליוותה אותי לאורך תקופת התואר.

אני מודה לטכניון על התמיכה הכספית הנדיבה בהשתלמותי.

תקציר

ארכיטקטורות מחשוב מודרניות מציבות אתגרים חדשים למפתחי תוכנה. כיום מעבדים מכילים יותר מליבה אחת, והמגמה היא לכלול מספר גדול יותר של ליבות בעתיד. מגמה זו מאלצת את מפתחי התוכנה לכתוב תוכנות יותר סקלביליות, שמסוגלות לנצל מספר גדול של ליבות על ידי שימוש במספר גדול של חוטים.

שימוש במספר גדול של חוטים יוצר מספר בעיות. גישה לזכרון משותף על ידי מספר גדול של חוטים מהווה צוואר בקבוק, ולכן על מפתחי התוכנה להימנע מכך. כמו כן, ההשהיה בגישה לזכרון מרוחק ארוכה יחסית לגישה לזכרון המטמון של המעבד, ולכן על המפתחים למזער את הגישה לזיכרון מרוחק וליצור תוכנה שהיא ידידותית-למטמון, כלומר תוכנה בה רוב הגישות הן גישות לזכרון המטמון.

מערכות רבות בעלי מעבדים-מרובים מגיעות כיום בתצורת NUMA, בתצורה זו לכל מעבד יש בנק זכרון משלו, אליו הוא ניגש דרך בקר הזכרון שלו. אם מעבד צריך לגשת לזכרון מרוחק, עליו לגשת אליו דרך חיבור מיוחד שמחבר אותו למעבדים מרוחקים. הגישה לזכרון מרוחק היא בעלת זמן השהייה גבוה יותר ורוחב הערוץ לזכרון מרוחק קטן יותר בהשוואה לגישה לזכרון שמחובר ישירות למעבד. לכן במערכות NUMA עדיף לגשת לזכרון המקומי על מנת לצמצם את התעבורה בין המעבדים.

בעבודה זו חקרנו שתי בעיות בהקשר של סקלביות ולוקליות.
העבודה הראשונה היא מימוש של מבנה נתונים של מאגר משימות עבור צרכנים-יצרנים. מבנה זה מיועד לשימוש על-ידי חוטים-יצרנים שמייצרים משימות, אותם החוטים-הצרכנים צריכים לבצע. החוטים-היצרנים מוסיפים משימות למאגר המשימות, והחוטים-הצרכנים לוקחים משימות מהמאגר. אם המאגר ריק, חוטים-צרכנים ידווחו שהוא ריק ולא יחזירו משימה. מבנה נותנים זה הוא מבנה נתונים בסיסי בתבנית התכנות של יצרן-צרכן שנפוצה בתכנות מקבילי.

המטרה שלנו בעבודה זו היא לתכנן ולממש מאגר משימות כך שהוא יהיה סקלבילי, מהיר וחסר-נעילות. אלגוריתם חסר-נעילות הינו אלגוריתם שמריץ מספר חוטים תוך הבטחה שאם חוט מבצע פעולות, אז אחד החוטים יסיים את הפעולה שלו בזמן סופי. כמו-כן האלגוריתם שלנו לא מבצע פעולות אטומיות חזקות כגון CAS ולא דורש פעולות סנכרון מסוג barrier, ברוב המקרים כשהמערכת מאוזנת.

מאגרי משימות כאלו לרוב מומשו ע"י תור FIFO בו סדר ההוצאה זהה לסדר ההכנסה. מכיוון שיש אילוץ על סדר ההוצאה, מבנים אלו דורשים סנכרון בעת הכנסת והוצאת איברים ולכן הם אינם סקלביליים. בעבודות מהתקופה האחרונה ויתרו על תכונת ה-FIFO לצורך סקלביליות. עבודות אילו אכן מציגות ביצועים טובים יותר מתורי FIFO, אך הן לא תמיד סקלביליות ביחס לינארי למספר החוטים, וכולן משתמשות בפעולות סנכרון במקרה הנפוץ, אפילו אם הוא לא נחוץ.

בעבודה זו תכננו ומימשנו מאגר משימות חסר-נעילות, סקלבילי וללא פעולות סנכרון במקרה הנפוץ. מבנה הנתונים מורכב ממערכת של מאגרים כמספר הצרכנים. לכל מאגר יש צרכן שהוא הבעלים שלו. כל מאגר תומך בפעולות של הוספת משימה, לקיחת משימה על-ידי הבעלים וגניבת משימה על-ידי צרכן שאינו הבעלים. צרכן שרוצה לקחת משימה מהמערכת תחילה ינסה לקחת משימה מהמאגר שלו, אם הוא נכשל ינסה לגנוב משימות ממאגרים של צרכנים אחרים. יצרן יכול להוסיף משימות לכל אחד מהמאגרים. המדיניות שלנו מודעת לארכיטקטורת המערכת, ובמערכות NUMA צרכנים מנסים לגנוב מצרכנים שנמצאים על אותו המעבד כמותם. יצרנים יעדיפו להוסיף משימות לצרכן שאיתו הם חולקים מעבד. מדיניות זו גורמת לכך שרוב הגישות יעשו לזכרון הקרוב לחוט ובכך תצמצם גישות למעבדים מרוחקים.

המאגרים של כל צרכן ממומשים עלי-ידי רשימות של משימות כאשר הרשימה מחולקת לנתחים וכל נתח מכיל מספר רשימות. לכל יצרן יש רשימה משלו, וכך אין צורך בסנכרון בין היצרנים. לכל נתח יש יצרן שהוא הבעלים של הנתח, לכל נתח יש שדה שמכיל את זהות היצרן שהוא הבעלים של הנתח. יצרן יכול לקחת משימות רק מנתחים שבבעלותו. כאשר יצרן מעוניין לגנוב נתח של יצרן אחר, הוא משנה את הבעלות של הנתח ואז מעביר אותו לרשימה שלו. כאשר נתח נגנב, יתכן שהבעלים המקורי כבר התחיל בפעולת לקיחה של משימה, במקרה זה הבעלים המקורי יבחין שהנתח נגנב וייקח את המשימה עם פעולה אטומית מסוג-CAS. מסיבה זו, כאשר יצרן גונב נתח, עליו לקחת את המשימה הבאה בעזרת פעולת CAS כדי למנוע מצב בו משימה נלקחת פעמיים. שיטה זו מאפשרת לנו לוותר על פעולות סנכרון כאשר אין גניבות, ומשתמשת בפעולות אלה רק כאשר יש צורך בהן.

על מנת לשמור על המאגרים מאוזנים מבחינת כמות המשימות, נקטנו בשיטה אותה אנו מכנים "איזון מונחה יצרנים". לכל צרכן יש מאגר של נתחים ריקים שפנויים לשימוש, כאשר יצרן זקוק לנתח חדש להוסיף לרשימת הנתחים שלו במאגר של יצרן כלשהו הוא לוקח נתח ממאגר הנתחים של אותו צרכן. אם מאגר הנתחים של אותו צרכן לא מכיל נתחים פנויים, היצרן מנסה להוסיף את המשימה שלו למאגרים של צרכנים אחרים. בנוסף, כאשר צרכן גונב נתח מצרכן אחר, נתח זה ישוב למאגר הנתחים של הצרכן הגנב לאחר שהוא יסיים לקחת ממנו את המשימות. באופן זה לצרכנים מהירים יהיו יותר נתחים פנויים ולצרכנים איטיים פחות נתחים פנויים וכך יצרנים יפנו יותר לצרכנים המהירים ופחות לאיטיים.

על-מנת להעריך את ביצועי המערכת, מימשנו אותה בשפת C++. בנוסף מימשנו גירסה של המערכת שבה הצרכן מבצע פעולת CAS גם כאשר אין גניבות, על מנת לראות את השפעת פעולות הסנכרון על המערכת. כמו כן מימשנו את אלגוריתם ה-ConcBags שפורסם ב-2011 ומייצג את האלגוריתם החדש ביותר המתחרה בנו. בנוסף מימשנו תור FIFO ומחסנית חסרי-מנעולים שמשתמשים בתשתית המערכת שלנו. בכל הבדיקות שערכנו המערכת החדשה הציגה ביצועים עדיפים על המערכות הקודמות וכמו כן היתה סקלבילית באופן לינארי למספר החוטים במערכת.

הבעיה השנייה שאותה חקרנו היא בעיית אחסון metadata (מידע על מידע) במערכת זכרון טרנזקציוני מבוסס תוכנה (Software transactional memory - STM). מערכות אלו

ממשות סביבת זמן ריצה שמאשרת למפתחי תוכנה לממש תוכנה מקבילית בקלות יחסית ללא שימוש במנעולים. במערכות TM, המתכנת מגדיר בלוקים שיבוצעו בצורה אטומית ומערכת ה-TM מריצה את הקטעים במקביל בצורה אופטימית. אם המערכת מזהה שתי טרנזקציות שניגשות לאותו מידע, היא תבטל את אחת מהן ותתחיל אותה מחדש.

חלק ניכר ממערכות אלו משתמש במנעולים על-מנת לוודא שטרנזקציות לא ניגשות לאותם תאים זכרון. הגישה הרגילה היא לאחסן מנעולים אלו בטבלת מנעולים רציפה בזכרון, אך לגישה זו יש מספר בעיות – מכיוון שטבלה זו משותפת, גם טרנזקציות שניגשות לאזורי זכרון שונים עלולות לגשת למיקומים קרובים בטבלה וכך להפריע אחת לשניה. בנוסף מכיוון שטבלה זו בגודל סופי וקבוע, יש מצבים בהם מספר מנעולים ממופים לאותה שורה וכך יתכן מצב שבו טרנזקציה תיכשל ללא סיבה.

פתרון אפשרי שהוצע למצב זה הוא לאחסן את המנעולים ביחד עם המידע. דרך זו מאפשרת גישה רציפה למידע ול-metadata וכך מעלה את הסבירות שהמידע יהיה במטמון. בנוסף בדרך זה אין התנגשויות בין פעולות שניגשות לאזורים שונים בזכרון ולכן הפתרון הוא יותר סקלבילי. לבסוף דרך זו מאפשרת נעילה ברמה של אובייקט לוגי גם בשפות שאינן מונחות אובייקטים.

בעבודה שלנו חקרנו את ההשפעה של לוקליות על הביצועים של מערכת STM. תחילה הרצנו מיקרו-בדיקות שבודקות את האפקט של לוקליות במערכות שונות. לאחר מכן מימשנו אב-טיפוס של מערכת STM מבוססת על TL2 בה ה-metadata נשמר עם המידע. על מערכת זו הרצנו בדיקות מאוסף הבדיקות STAMP המיועדת לבדיקת מערכות STM והראנו את ההשפעות של אחסון ה-metadata על זמן הריצה ביצועי זכרון המטמון ופרמטרים נוספים.