LUTS: A Lightweight User-Level Transactional Scheduler

Daniel Nicacio *Alexandro Baldassin*
Guido Araujo

Technical Report - IC-10-33 - Relatório Técnico

December - 2010 - Dezembro

LUTS: A Lightweight User-Level Transactional Scheduler

Daniel Nicacio Alexandro Baldassin Guido Araujo

Abstract

Transaction scheduling techniques have been used as a way to reduce aborts by avoiding conflicting transactions to run in parallel, while improving core usage. Traditional scheduling techniques are based on the operating system preemptive scheduler, and thus have little or no control in choosing the best transactions to execute. In this paper, we propose LUTS, a Lightweight User-Level Transaction Scheduler. Unlike other techniques, which serialize conflicting transactions, LUTS provides the means for selecting another transaction to run in parallel, thus improving system throughput. Moreover, it avoids most of the issues caused by pseudo parallelism, as it only launches as many system-level threads as the number of available processor cores. This is achieved by maintaining a set of *execution context records* (ECRs) which are used to encapsulate the state of the threads. ECRs are inserted into the scheduler queue and are selected for execution, according to its conflicting history with the running transactions. We discuss LUTS design and present a prototype implementation and integration with a state-of-the-art STM, tinySTM. Experimental results, conducted with the STAMP benchmark suite, show LUTS efficiency when running high contention applications, and its stability as the number of threads increase. LUTS achieved better performance results than existing techniques in every STAMP benchmark program, executing programs up to 1.69 times faster.

1 Introduction

The emergence of multicore processors has renewed the interest in concurrent programming and effectively moved it into mainstream. During the last few years, extensive research has been carried out into new programming models and abstractions, of which transactions (or atomic blocks) is certainly a promising one [11]. A transaction can be considered as an atomic block of code that is executed in isolation from the rest of the system, offering a convenient synchronization mechanism. One of the major advantages of transactions relies on the fact that it moves most of the burden from the programmer to the underlying transactional memory (TM) subsystem (similar to programming with a global lock).

While transactions perform well on workloads with reasonable amounts of parallelism, they may substantially degrade performance on those which exhibit higher data contention. In such cases, two or more transactions tend to compete for the same data and, since at least one of them updates the referred data, a *conflict* occurs. Deciding what to do when a conflict arises is the responsibility of the so called *contention manager* [10], an important component of any TM system. Usually, a conflicting transaction is aborted and restarted after some time. Although certain contention policies may provide good performance for some workloads, it can also considerably downgrade performance for others.

Contention management is a central issue when the system is characterized by *pseudo parallelism* [5]. Pseudo parallelism occurs when the total number of system-level threads is greater than the available number of processor cores, forcing several threads to share the same core. In such scenario, performance is greatly degraded as: (i) more transactions running simultaneously naturally increase the likelihood of conflicts; (ii) same-core transactions are more likely to conflict to each other, specially when they are longer than the scheduler quantum; and (iii) more threads sharing the same core tend to cause more cache misses and page faults. Dealing with the pseudo parallelism is a central problem in the design of efficient contention managers.

Until recently, the research on contention managers focused mainly on devising new contention policies in the hope to increase performance [19, 9, 20]. However, typical contention managers lack the knowledge necessary to effectively increase transaction throughput, since they only take actions after the conflict has occurred. They can use different techniques to select which transaction to abort and the amount of time to delay the restart, but they cannot prevent two conflicting transactions from executing concurrently. Moreover, the benefits are not easy to predict as there is no easy way to decide whether a given delay is appropriate: it might be too short and cause later conflicts, or too long and do not exploit concurrency to its fullest. As a result, research attention has shifted to *scheduling-based contention management* [22, 5, 1, 7, 2, 13].

In the scheduling approach, questions such as when to start a transaction or whether two transactions should run concurrently are taken into account. Ideally, we would like to avoid starting two transactions that will conflict in the future. Earlier works [22, 5, 1] have proposed serialization as the main contention management mechanism: they keep track of the likelihood of a transaction to abort and, when its conflict probability reaches a given threshold, they serialize such transaction. Hence, in a high contention scenario, transactions with repeated aborts will be serialized one after another, thus reducing the total number of conflicts and wasted work. Later proposals [7, 2] have suggested a proactive approach, wherein the decision of whether to start a given transaction is taken before the transaction starts executing. Such decision is usually based on the transaction past conflict history and may prevent a conflict to happen beforehand. Serialization can also be employed in case a transaction is not eligible to immediate execution. Most of the scheduling-based contention management proposals are implemented at the user-level and rely on costly synchronization primitives such as locks and condition variables, which invariably involve system calls. A recent work [13] have proposed kernel-level scheduling support in order to reduce the overhead presented in the previous user-level approaches, at the cost of changing the OS scheduler.

In this paper we propose LUTS, a Lightweight User-level Transactional Scheduler. LUTS implements a fully cooperative scheduler, and does not rely on the system-level scheduler for context switching the transaction threads. LUTS novel user-level cooperative approach presents two main advantages when compared to state-of-the-art TM schedulers. First, it deals with the pseudo parallelism issue in an elegant way, by only spawning as many system-level threads as the number of available processor cores and handling the exceeding threads internally. Second, LUTS allows the TM subsystem to efficiently access the runnable transaction queue and switch the execution to any of them. This allows the design of more precise proactive scheduling schemes, not possible with the current approaches, which are restricted to either serialization or yielding.

More precisely, the contributions of this paper are as follows:

- A cooperative scheduler (LUTS), designed with the goal of eliminating pseudo parallelism. It creates at most as many system-level threads as available processor cores, transparently handling the exceeding threads.
- A novel proactive conflict-avoidance heuristic that uses LUTS scheduling capabilities to avoid starting transactions that are likely to conflict. When a transaction is about to start, we check the probability of conflict among executing transactions. If this probability is high, we choose a transaction that is less likely to abort from LUTS runnable queue.
- A prototype implementation of LUTS integrated into a state-of-the-art software transactional memory (STM), tinySTM [8].
- An evaluation of LUTS approach to contention management through the STAMP [17] benchmark suite. In general, the experimental results show LUTS efficiency in dealing with pseudo parallelism issues. When compared to earlier approaches such as [22] and [7], LUTS presented the best overall performance. We also show results of our proactive conflict-avoidance heuristic.

The rest of the paper is organized as follows. Section 2 presents an overview of LUTS. Section 3 discusses an implementation for x86 architectures, while Section 4 assesses the implementation performance using STAMP applications. Finally, Section 5 describes prior works and Section 6 concludes.

2 Overview

The design of LUTS is based on two main assumptions: (1) the number of system-level threads (henceforth referred to as SLTs) should not exceed the number of available cores, and (2) an STM system would benefit from advanced scheduling capabilities. Assumption (1) aims at reducing pseudo parallelism issues, while assumption (2) is guided by the idea that more robust conflict-avoidance mechanisms can be created if scheduling details are exposed to the STM system. Throughout this section we elucidate these two key design points.

Figure 1 shows a general overview of a LUTS-based system. It sits on top of the operating system and provides a threading interface to applications and a scheduling interface to the STM library. Hence, LUTS can be seen as an abstraction to threads and scheduling services. Notice that the corresponding OS services are not explicitly used by the application and STM library. Table 1 presents a summary of the main routines provide by the LUTS interface. We describe each one in more detail in the following sections.

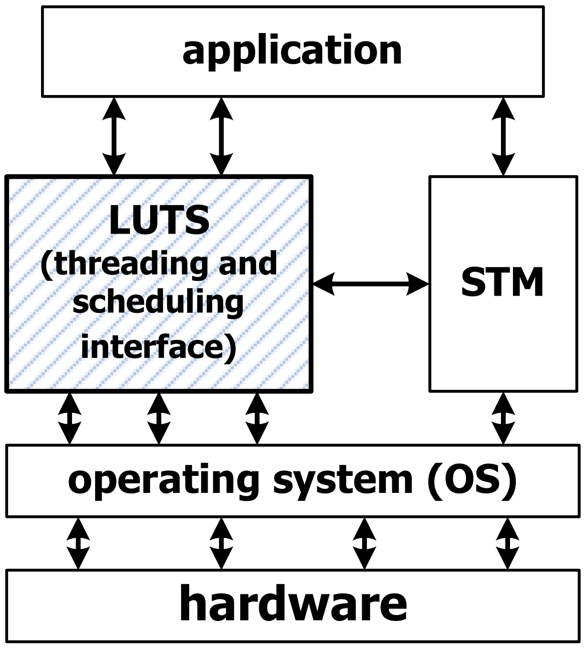


Figure 1: LUTS provides a threading and scheduling interface to applications and STM libraries.

2.1 Threading interface

LUTS provides the basic interface to a single program multiple data (SPMD) style of parallel programming [3]. This includes a routine to create a certain number of working threads (`luts_init`) and the ability to instruct all created threads to start executing the same program (`luts_start`). Notice that no synchronization support (locks and condition variables) is provided, since we are assuming a transactional model implemented by some STM. However, there is one important exception: barrier synchronization. Given that transactions are not able to implement such a construct, LUTS does supply an interface (`luts_barrier_wait`).

A key distinction, which differs LUTS from other conventional approaches, is that it does not automatically create as many system-level threads as required by a call to `luts_init`. Instead, it creates a number of *execution context records* (ECRs), which encapsulate the state of a thread. Each ECR is inserted into the scheduler runnable queue and is eligible for later execution. LUTS only creates as many system-level threads as the number of available cores. The dispatcher is responsible for taking an ECR from the runnable queue and mapping it to a system-level thread, as illustrated by Figure 2.

The main reason to adopt LUTS' threading design is to overcome the potential shortcomings caused by pseudo parallelism. In a conventional system, multiple transactions may run on the same processor core: the OS scheduler can preempt a thread that is running a transaction at any time and switch to another thread running a different transaction. As pointed out by earlier works [5, 13], this behavior can dramatically increase the probability

Threading interface	
Routine	Description
luts_init(`num_threads`)	Initialize the system with the specified number of threads
luts_start(`func, args`)	After the call, each thread will start executing `func` with arguments `args`
luts_barrier_wait()	Execution only proceeds after all threads have arrived at the barrier
luts_shutdown()	Clean up LUTS state and exit

Scheduling interface	
Routine	Description
luts_yield()	Current thread is inserted into the tail of the runnable queue, and another one is taken from the head for execution
luts_setid(`txId`)	This routine is used to set the transaction identifier for the current execution context
luts_switch(`txId`)	LUTS will look up its running queue for a transaction with an ID different of `txId`. If found, context is switched from current thread to the selected one

Table 1: Summary of LUTS threading and scheduling interface routines.

of conflicts and decrease overall performance. Furthermore, pseudo parallelism also tends to increase the number of cache misses and page faults. LUTS design aims at executing at most one transaction thread per core. This is enforced by: (i) creating at most as many system-level threads as the number of available cores, and (ii) setting the thread affinity to a specific core. This practically avoids having the same core executing more than one transaction thread. Of course, the OS scheduler can still preempt a thread executing a transaction if other applications are executing concurrently. In this case, we can decrease the number of LUTS spawned threads. In this paper, we assume no other applications are executing in parallel and leave the investigation of such scenarios to future work.

2.2 Scheduling interface

As previously mentioned, in LUTS a thread is represented by an ECR. When the number of ECRs is equal or less than the number of threads spawned, the behavior of the system is similar to conventional systems: each ECR is continuously mapped to a single SLT. When the number of ECRs is greater than the available SLTs, the dispatcher selects an ECR and maps it into an SLT for execution. After an ECR is mapped and starts executing, there are only two ways for the corresponding SLT to become free again: either the work is finished or the ECR voluntarily relinquish control by calling, for example, `luts_yield`. Hence, LUTS employs a cooperative approach to scheduling.

There are two main advantages in adopting a cooperative over a preemptive scheduler in the context of this paper. First, it is very simple to implement and efficient. Second, coordination avoids the risks caused by pseudo parallelism and preemption as discussed

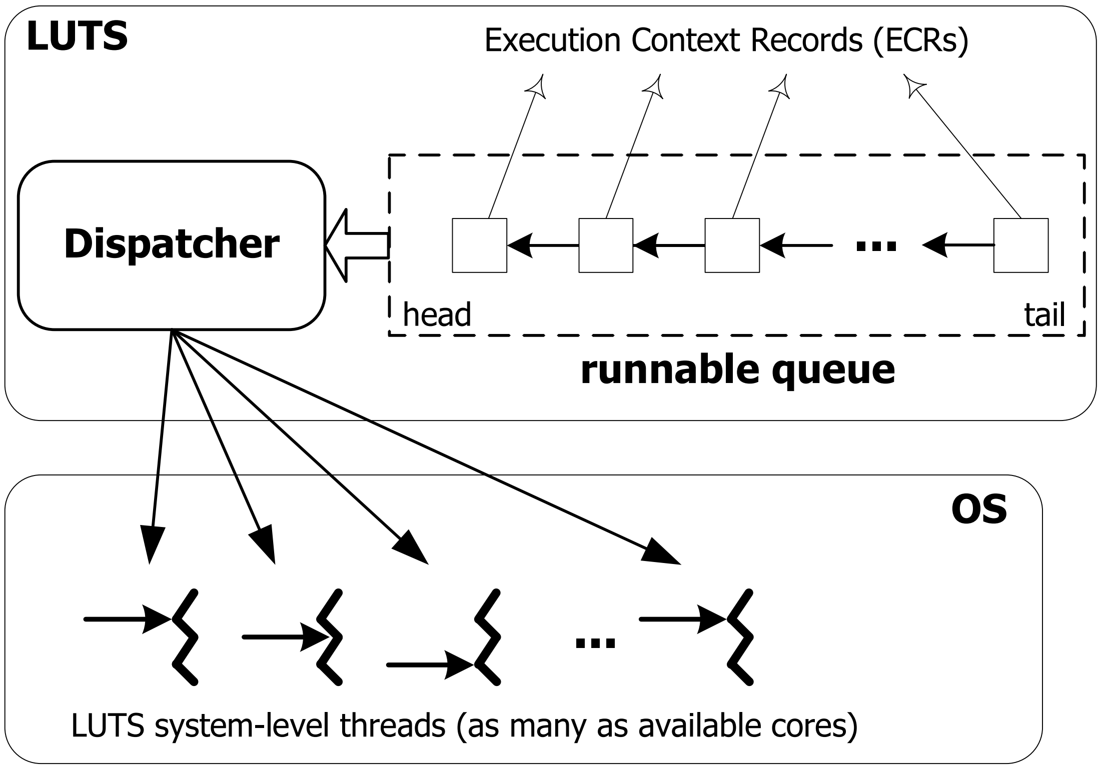

LUTS
Execution Context Records (ECRs)
Dispatcher
head
tail
runnable queue
OS
LUTS system-level threads (as many as available cores)

Figure 2: LUTS thread mapping. The dispatcher selects an ECR from the running queue and maps to a system-level thread.

in Section 2.1. Therefore, we avoid blocking a thread that is running a transaction and potentially decrease the probability of inter-transaction conflicts. Notice that the types of applications we are expecting to run are not IO-bound (IO operations inside transactions pose some difficulties [14, 21]), and therefore a preemptive scheduler is not really necessary. Moreover, the application code does not need to cope with scheduling issues, such as yielding the SLT, as the STM library transparently deals with that, by means of the LUTS interface.

The basic interface to our scheduler is `luts_yield`, with a similar semantic to the `sched_yield` routine provided by UNIX-like operating systems. This routine forces the current context to relinquish the SLT. The context is inserted into the tail of the scheduler runnable queue, and another one is taken from the head and executed by the dispatcher. The remaining routines allow STM libraries to create advanced conflict-avoidance heuristics. LUTS maintains internally, for each ECR, a field with the purpose of identifying a transaction. STM libraries can set this field via the `luts_setid` routine. Additionally, STMs can exert influence on the default scheduling policy by calling `luts_switch`. The argument `txId` passed to this routine instructs LUTS to find an ECR from its runnable queue with an identification field different of `txId`. If such an ECR is found, the execution is switched to the new ECR. Otherwise, nothing happens. We elaborate on these capabilities and present a new conflict-avoidance heuristic in Section 3.3.

Listing 1: LUTS metadata

```
1  typedef struct execution_context_record {
2    int  id;
3    int  txId;
4    int  barrier_sense;
5    word *sp;
6    word stack[STACK_SIZE];
7  } ecr_t;
8
9  /* global state */
10 static concurrent_queue runnable_queue;
11
12 /* thread local (SLT) variables */
13 static __thread ecr_t *current = NULL;
14 static __thread word *stack_pointer = NULL;
```

3 Implementation

In this section we detail LUTS prototype implementation, briefly describe how to integrate it with typical STM libraries, and develop two conflict-avoidance heuristics based on LUTS scheduling capabilities. For the sake of presentation we opted for showing simplified rather than optimized code. We loosely follow a C-like syntax, but allow for extensions in order to facilitate the presentation.

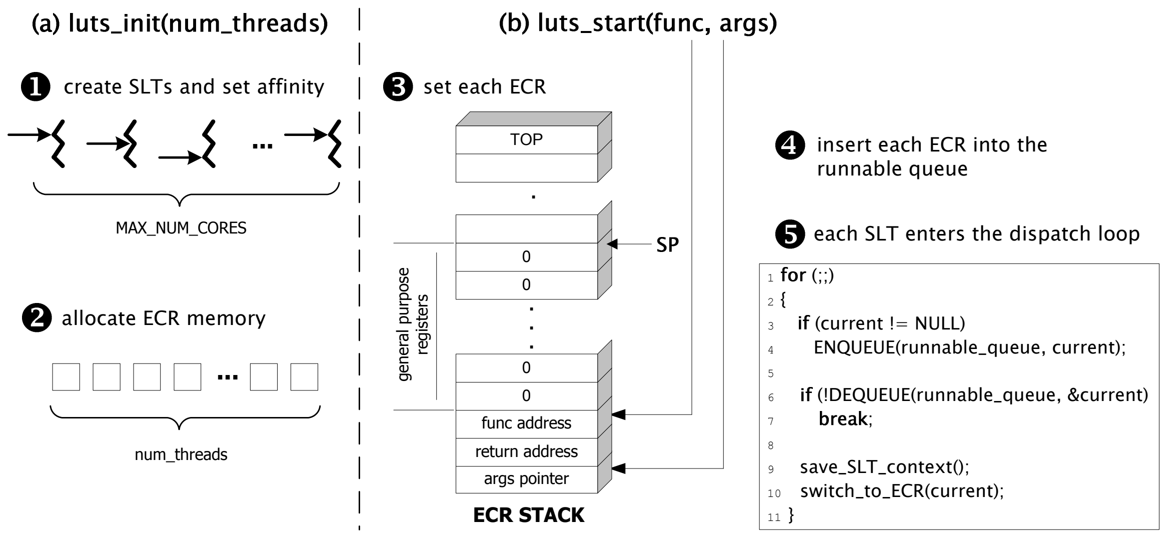
3.1 LUTS

The prototype implementation targets a Linux-based system and was mostly developed using the C programming language. Assembly language was required for certain parts of the code, particularly those dealing with context switching. In that case, we assume a x86-like processor and use the AT&T syntax for assembly coding.

3.1.1 Metadata

The main global metadata maintained by our prototype appears in Listing 1. At the core of LUTS is the execution context record, defined in lines 1 to 7. It stores the ECR and transaction identifiers (lines 2 and 3), barrier metadata (line 4, discussed later), the stack pointer and the stack itself (lines 5 and 6). All information necessary to resume execution of a context is stored into the stack, including the contents of the general purpose registers and the program counter. The stack pointer is stored separately in the `sp` field, since upon resuming we need to locate the position within the stack that contains the state saved by the suspended context.

ECRs are maintained in a shared runnable queue (line 10) by the scheduler. Our current implementation uses the two-lock concurrent queue proposed by Michael and Scott [16]. Finally, each SLT maintains a pointer to its current assigned ECR (line 13) and a stack pointer holding the address of the SLT original stack (line 14). When an ECR is put to


(a) luts_init(num_threads)
❶ create SLTs and set affinity
MAX_NUM_CORES
❷ allocate ECR memory
num_threads
(b) luts_start(func, args)
❸ set each ECR
TOP
SP
general purpose registers
0
func address
return address
args pointer
ECR STACK
❹ insert each ECR into the runnable queue
❺ each SLT enters the dispatch loop

```
for (;;)
{
  if (current != NULL)
    ENQUEUE(runnable_queue, current);

  if (!DEQUEUE(runnable_queue, &current)
    break;

  save_SLT_context();
  switch_to_ECR(current);
}
```

Figure 3: LUTS operation. In (a), the SLTs and ECRs are created. In (b), each ECR is set, enqueued into the runnable queue, and each SLT starts executing the dispatch main loop.

execute on an SLT, the SLT first saves its state onto the stack and only then switches to the specified ECR. Had we not saved the stack pointer, we would not be able to return the control to the SLT main code.

3.1.2 Operation

LUTS operation can be broadly divided into two main phases: initialization and execution. Figure 3 illustrates the main substeps in each phase. The first task of the initialization routine is to create the system-level threads and set their affinities accordingly ❶. As previously discussed, LUTS never launches more threads than available cores. The default behavior is to create an SLT for each processor core and configure the affinity to that core so as to get the maximum performance. In the initialization phase we also allocate dynamic memory for the ECRs ❷. It is important to notice that SLTs are only created at this point. They are kept on hold till after the system starts.

The execution stage is invoked by a call to `luts_start`. A pointer to the function to be executed (`func`) needs to be provided, along with optional arguments (`args`). Initially, each ECR stack is set up using the layout showed in step ❸ of Figure 3 [1]. At the bottom of the stack lies the arguments' pointer, followed by the return address, function address, and the contents of the general purpose registers (initially zero). The ECR stack pointer (`sp`) is then set to the stack address of the last general purpose register value. The stack is organized this way because the context can be easily switched to an ECR: it suffices to restore the stack pointer register (via the `sp` field), pop the contents of the general purpose

[1]Recall that the current implementation is assuming an x86 machine. The actual stack organization may vary for different architectures.

registers, and issue a `ret` instruction. The PC is then set to the value of the function address, appropriately transferring control to the desired task. At this point, notice that the stack layout adheres to the default calling convention for x86 `C` compilers (`cdecl`): at the top is the return address (which is set to an internal LUTS procedure that correctly finalizes the context), and one address deeper is the arguments' pointer. After an ECR is set up, it is enqueued into the runnable queue ❹ and eligible for execution.

Finally, after all ECRs are initialized and inserted into the runnable queue, the SLTs enter the dispatcher loop ❺. Initially, each SLT has no ECR assigned (`current == NULL`), and the test at line 3 fails. Otherwise the current ECR would be inserted at the tail of the queue (line 4). An ECR is dequeued from the runnable queue at line 6. If there is no available job, the SLT exits the loop (line 7) and is finalized. With the ECR assigned, the state of the SLT is saved (line 9) and the context switched to the ECR (line 10). From the presented code we can see that the dispatching operation is not centralized. This is important since a centralized approach might require extra synchronization, increasing the contention on the scheduler.

Not shown in Figure 3 are the remaining operations for yielding an SLT and switching to an ECR based on a transaction identifier. The former operation is shown in Listing 2. The first step in yielding is to save the current ECR state into its stack, which is performed by the assembly instructions from line 3 to 10. The stack pointer is saved at line 11. Later, when this ECR resumes, it will continue execution from line 26. After the ECR state is saved, the SLT context is restored (lines 15 to 24), and control is returned to the main dispatch loop as presented in step ❺ of Figure 3. The switch operation works similarly to yield, but we make sure that when the dispatch code is executed, it looks for a specific ECR in the runnable queue, instead of just selecting the head element.

3.1.3 Barrier synchronization

The last important aspect of our current prototype concerns barrier synchronization. We cannot just use the native barrier support (for instance, the one provided by pthreads) out of the box, since there are usually more ECRs than SLTs. Therefore, we implemented our own barrier support based on the sense-reversing algorithm proposed by Mellor-Crummey and Scott [15].

Listing 3 shows the barrier data type (lines 1 to 4), followed by the barrier declaration (line 6). Our sense-reversing implementation, a direct implementation of the original algorithm, is shown next (lines 8 to 20). Each ECR stores a barrier field (`barrier_sense`) which is reverted every time a barrier is entered (line 10). When an ECR process enters the barrier, the barrier counter is atomically decremented (line 12) and, if it is the last process to arrive at the barrier, it resets the counter (line 14) and reverts the global sense variable (line 15). Otherwise the process will remain in busy wait (line 18) until all processes arrive at the barrier. An important modification from the original algorithm is that we require the ECR process to perform a yield operation (line 19), thus allowing all ECRs to eventually enter the barrier and make progress.

Listing 2: Yield operation

```c
void luts_yield()
{
  asm volatile ( "pushl $resume_point" "\n\t"
                 "pushl %%eax"  "\n\t"
                 "pushl %%ecx"  "\n\t"
                 "pushl %%edx"  "\n\t"
                 "pushl %%ebx"  "\n\t"
                 "pushl %%esi"  "\n\t"
                 "pushl %%edi"  "\n\t"
                 "pushl %%ebp"  "\n\t"
                 "movl %%esp, %0" "\n\t"
                 : "=m"(current->sp));

  /* restore SLT context */
  asm volatile( "mov %0, %%esp" "\n\t"
                "popl %%ebp"    "\n\t"
                "popl %%edi"    "\n\t"
                "popl %%esi"    "\n\t"
                "popl %%ebx"    "\n\t"
                "popl %%edx"    "\n\t"
                "popl %%ecx"    "\n\t"
                "popl %%eax"    "\n\t"
                "ret"           "\n\t"
                ::"m"(stack_pointer));

  asm volatile ("resume_point:");
}
```

Listing 3: Barrier implementation

```c
struct luts_barrier {
  unsigned int count;
  int sense;
} lbarrier_t;

static lbarrier_t ecr_barrier;     /* ECR barrier */

void luts_barrier_wait()
{
  current->barrier_sense = !current->barrier_sense;

  if (ATOMIC_FETCH_DEC_FULL(&ecr_barrier.count) == 1)
  {
    ecr_barrier.count = num_ecrs;
    ecr_barrier.sense = current->barrier_sense;
  }
  else
    while (ecr_barrier.sense != current->barrier_sense)
      luts_yield();
}
```

3.2 STM integration

STM libraries will interface with LUTS scheduling services. All that is required is to include the scheduler header file and invoke the appropriate routines. Some STM implementations store the transaction descriptor in thread local storage (TLS) to avoid passing it as a parameter to every transactional primitive routine. This is the case, for example, with tinySTM [8]. For such cases, LUTS provides wrapper routines to simulate thread private memory with an interface similar to pthreads (create key, set specific, and get specific). Moreover, since the access to the transaction descriptor is critical, LUTS provides corresponding set and get routines. We also perform function inlining in order to reduce the overhead in accessing the transaction descriptor. For implementations that do not use TLS, such as the TL2 [4] implementation provided with the STAMP benchmark, there is no need to use the wrapper routines.

3.3 LUTS-based conflict-avoidance heuristics

This section presents two heuristics built around LUTS scheduling capabilities, `CILUTS` and `CTLUTS`. In both heuristics, we first determine proactively the conflict probability of starting a transaction and invoke LUTS to switch the execution to a different transaction when possible, thus avoiding serializing the execution. Evaluation results for both techniques are discussed in section 4.

3.3.1 `CILUTS`

In the first approach, `CILUTS`, each transaction maintains internally a variable describing its conflict probability. To quantify this variable we use the concept of contention intensity (CI), firstly described by Yoo and Lee [22] in their Adaptive Transaction Scheduling (ATS) proposal, and given by the following equation:

$$CI_n = \alpha \times CI_{n-1} + (1 - \alpha) \times CC$$

Initially, CI is set to zero, and this equation is evaluated at each commit or abort operation. The current contention (CC) is set to 0 on commits and to 1 on aborts. The α value determines which time window we prioritize: the past history or the current contention information. After testing different α values (0.3, 0.5 and 0.75), 0.75 turned to be the most effective for STAMP benchmark, so we set α to 0.75, giving a little more weight to the past history.

Our technique differs from ATS when we find that a transaction is likely to abort. Instead of serializing the transaction, like ATS does, we use LUTS to find a more appropriate transaction to take its place, by using the interface method `luts_switch(txId)` to switch the current transaction to another one with a different ID. By doing so, we can reduce the number of conflicts and do not waste processor time waiting for a transaction to commit. Therefore we increase the application parallelism and improve its performance.

3.3.2 `CTLUTS`

`CTLUTS` uses a conflict table to keep track of the conflict probabilities among transactions. The pseudo code for the heuristic is presented in Listing 4. Besides the conflict table itself (line 1), we also maintain a vector of active transactions (line 2). For each processor core we keep in this vector the ID of the transaction it is currently executing. The conflict table is made as large as the number of different transactions in the application. Transactions are differentiated by the address of their first instruction. When the transactional system is initialized, we reset the conflict table (lines 5 and 6) and set all entries in the active vector to invalid (lines 8 and 9).

Since `CTLUTS` adopts a proactive approach, the most important part of the heuristic takes place upon the start of a transaction (lines 11 to 22). For each valid transaction ID in the active vector, we check the conflict probability between the transaction pair and, if above a certain threshold (line 17), we invoke LUTS switch routine (line 18) to choose another transaction. The main idea of the heuristic is to avoid starting a transaction that is doomed to abort and reduce system contention. Ideally, a transaction with better chances to commit will be chosen from the runnable queue by LUTS. If there is no such a transaction then the current transaction is started anyway. Therefore, the worst case scenario for the heuristic happens when no transaction from the runnable is eligible to execute (its ID is the same of the current one). When the transaction is successfully initiated, it sets the corresponding entry in the active vector to its ID (line 22).

The conflict table is updated when a transaction fails (lines 24 to 32) or commits (lines 34 to 42). Both cases present a similar behavior: the ID of the transaction executing in each concurrent thread is retrieved from the active vector and, if it is different from invalid, the conflict probability of the transaction pair is increased in case of an abort (line 31), or decreased in case of success (line 41). It is worth noticing that the accesses to the active vector and the conflict table are not explicitly synchronized, therefore allowing data races. For instance, when we scan the active vector to update the conflict table, the read access can yield stale values since another thread might be writing to the vector. We consider these data races benign: they add an imprecision to the heuristic prediction but they cannot cause any execution fault. For `CTLUTS`, it is preferable to incur some imprecision than paying the high cost of synchronization. The same approach was adopted by Blake et al. [2] in a similar context.

4 Experimental Results

In this section we seek to investigate the performance of our prototype implementation of LUTS and the two proposed heuristics using tinySTM 1.0.0 as the base STM system. We conducted the experiments on an Intel Xeon E5405 with two 2.0GHz quad-core processors (8 cores in total), a 12MB L2 cache, and 4GB of RAM. The machine runs a typical Linux distribution with kernel version 2.6.32-25. All applications were compiled using gcc version 4.4.3, with optimization flag `-O3`.

The evaluation was done using programs from the STAMP 0.9.10 benchmark [17]. STAMP includes a gene sequencing program (Genome), a bayesian learning network (Bayes),

Listing 4: CTLUTS

```
double conflictTable[][];
int activeTx[];

upon stm_init
  for each transaction pair (t1, t2)
    resetCT(t1, t2);

  for each core i
    activeTx[i] = INVALID;

upon start
  for each core i {
    int other_id = activeTx[i];

    if (other_id == INVALID) continue;

    if (ProbCT(tx_id, other_id) > threshold) {
      schd_switch(tx_id);
      break;
    }
  }
  activeTx[thisCore] = tx_id;

upon abort
  activeTx[thisCore] = INVALID;
  for each core i {
    int other_id = activeTx[i];

    if (other_id == INVALID) continue;

    increaseProbCT(tx_id, other_id);
  }

upon commit
  activeTx[thisCore] = INVALID;
  for each core i {
    int other_id = activeTx[i];

    if (other_id == INVALID) continue;

    decreaseProbCT(tx_id, other_id);
  }
```

a network intrusion detection algorithm (Intruder), a k-means clustering algorithm (KMeans), a maze routing algorithm (Labyrinth), a set of graph kernels (SSCA2), a client-server reservation system simulating SpecJBB (Vacation), and a Delaunay mesh refinement algorithm (Yada). The input data set used was the largest one suggested by the STAMP authors. Programs KMeans and Vacation have two variations each, one for low and other for high contention level.

We report results for 6 different configurations: a baseline implementation and 5 different scheduling techniques. All configurations use the same base code, providing fairness in our comparisons. More specifically, the following configurations are used:

- **Original**: the baseline tinySTM implementation (version 1.0.0). For the experiments we configured tinySTM with the write-back and encounter-time locking (ETL) strategy. For comparison, we adopted the `CM_SUICIDE` contention policy, which immediately restarts a transaction on abort.

- **ATS**: an implementation of Yoo and Lee adaptive transaction scheduling technique [22]. We use a single global queue for all transactions, as suggested by the authors. Before running the experiments reported here, we conducted a sensibility study on α with values 0.3, 0.5 and 0.75, and on threshold with values 0.3, 0.5 and 0.7. We found out that the combination of 0.75 for α and 0.5 for threshold resulted in best overall performance, and thus was adopted in the experiments.

- **Shrink**: an implementation of the Shrink scheduler proposed by Dragojevic et al. [7] and also integrated with SwissTM [6]. The code was taken from the authors website [2] for tinySTM version 0.9.5, and adapted to the current version (1.0.0) by us. We did not change the parameters in the code: succ_threshold = 0.5, locality_window = 4, confidence_threshold = 3, `c1` = 3, `c2` = 2, `c3` = 1.

- **LUTS**: the scheduler with a fixed circular order scheduling policy (round-roubin). The scheduling interface is not used by the STM code in this configuration, allowing us to measure the gains of our approach in pseudo parallelism scenarios.

- `CILUTS`: the heuristic using contention intensity and LUTS scheduling interface as discussed in Section 3.3.1. Similarly to ATS, we adopted $\alpha = 0.75$ and threshold $= 0.5$ in the experiments.

- `CTLUTS`: the heuristic using the conflict table to track conflict probabilities and LUTS switching feature, as explained in Section 3.3.2. In the experiments we use a threshold value of 0.5.

For every application and a specific configuration, we report the average over 10 runs in order to reduce the variance in the results. Even so, we noticed a high variance for the Bayes application and omitted it. We also ommited the results for SSCA2 since it presented a livelock scenario in all configurations.

[2]http://lpd.epfl.ch/site/research/tmeval

4.1 Speedup

We show speedup results for the 6 configurations discussed previously with respect to the baseline tinySTM with a single thread. Figure 4 shows the speedup comparison for eight STAMP programs when the number of threads is varied from 1 to 64.

If we consider LUTS performance when the number of threads does not exceed 8, no important overhead can be noticed relative to the base STM. In fact, LUTS even outperformed the base STM and the considered heuristics in Labyrinth with 8 threads. From 16 to 64 threads, LUTS achieves the same performance as running 8 threads and, unlike the original implementation, it sustains this performance as the number of threads increase; this behavior is consistently maintained for every STAMP program. With 64 concurrent threads we can notice a speedup of 3.9x for KmeansHigh, whereas ATS and Shrink reached about 2.9x and 2.6x, respectively. LUTS also surpasses the other heuritics in the case of the Intruder application.

Besides the gains achieved with LUTS alone, we also notice a performance boost in some applications due to the heuristics `CILUTS` and `CTLUTS`. In programs with larger transactions and more transaction options, like Labyrinth and Yada, a good performance improvement was achieved. While LUTS achieved speedups of 3.9x and 1.8x in these programs (64 threads), `CILUTS` and `CILUTS` managed to speed up Labyrinth by 4.6x and Yada by 2x. In fact, for Labyrinth, `CTLUTS` and `CILUTS` also performed better than ATS and Shrink.

In programs with short transactions and few different transactions, like Intruder, KmeansHigh and KmeansLow, LUTS-based heuristics generated a slight overhead and did not improve LUTS performance. Specially for Kmeans, they were surpassed by ATS and Shrink. The source of the overhead for both `CILUTS` and `CTLUTS` comes from the extra actions performed during the transaction start, commit, and abort operations. Therefore, the overhead is proportionaly larger in programs with short transactions, making programs with large transaction better candidates for those heuristics. Moreover, avoiding the abort of a large transaction is more effective than avoiding the abort of a short one.

In general, we can conclude that LUTS delivers good performance on overloaded scenarios no matter how many threads are launched at the application startup, effectively dealing with pseudo parallelism issues. Moreover, the ability to switch to another transaction instead of applying serialization also can pay off, as we observed in the Labyrinth application.

4.2 Aborts

As previous work stated [18], some programs spend a good amount of time on aborted transactions; this can reach up to 22% for the application Labyrinth with 8 concurrent threads. Our scheduling policies manage to avoid such aborts and execute a non-conflicting transaction instead of a doomed transaction. By doing so, we achieve performance improvements as shown in Figure 4. Figure 5 shows the abort ratio in each STAMP program for each of the techniques studied in this work. This information allow us to better understand the effects of aborts on programs using STM. For example, applications with longer transactions benefit more when aborts are avoided than applications with short ones.

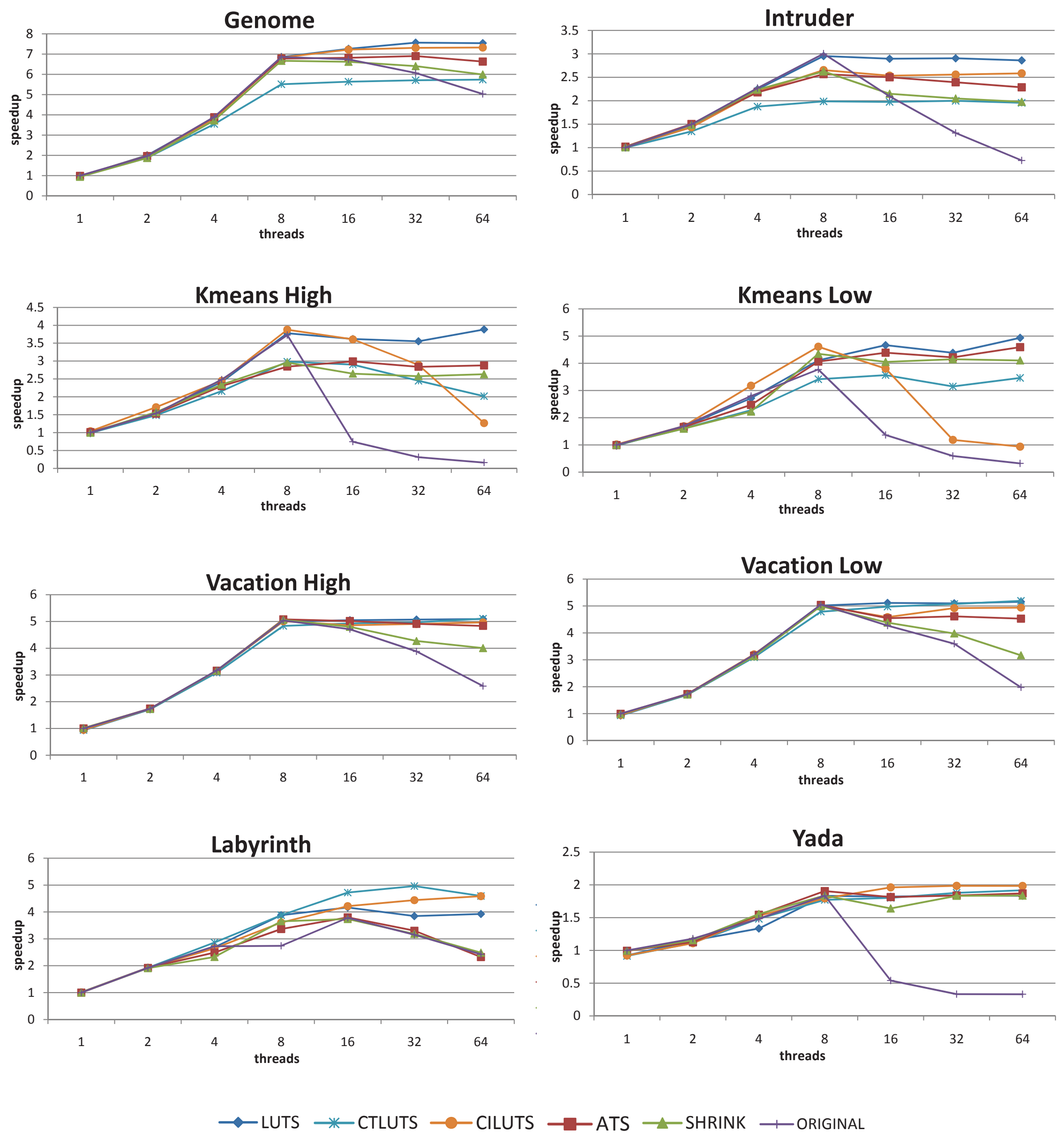


Figure 4: Speedup achieved on STAMP benchmark.

LUTS does not put any effort on avoiding conflicts between threads. However, by limiting the amount of concurrent threads to the number of available cores, it also limits the number of conflicting transactions. As expected, Figure 5 shows that from 8 to 64 threads, the amount of aborts with LUTS is steady in all programs and is always lower or equal than the number of aborts caused by the base STM system.

When scheduling policies `CILUTS` and `CTLUTS` are introduced, we expect to see the number of aborts to decrease even more. In fact, a reduction can be seen for practically all heuristics. Although the number of aborts decreased, this reduction is not necessarily reflected on program speedups. For STAMP, the abort reduction in Yada and Labyrinth are translated in speedups of 1.08x and 1.17x, when compared to LUTS. These programs use long transaction and a single abort has a high cost, so eliminating these aborts improved the overall performance. On the other hand, Intruder did not have a performance improvement despite the reduction on the number of aborts, as its transactions are short, generating low abort costs.

5 Related Work

STM contention management has been primarily researched in the context of modular contention managers, introduced by Herlihy et al. [12] for obstruction-free STM implementations. Due to its modular nature, a plethora of contention policies [10, 9, 19, 20] have been devised with the purpose of decreasing the number of conflicts and enabling system progress. Lock-based implementations have usually employed simpler heuristics, such as aborting the conflicting transaction and delaying the restart by using an exponential back-off mechanism. Despite the progress on contention managers, STM systems have not been able to anticipate conflicts and increase system throughput [22, 13]. On the contrary, traditional contention managers use a reactive (damage-control) strategy, instead of focusing on avoiding conflicts in the first place. Recently, the focus has shifted to scheduling-based contention management as discussed next.

Yoo and Lee [22] describe the first scheduling mechanism, adaptive transaction scheduling (ATS), aimed at reducing transaction re-execution in highly contended scenarios. Each transaction maintains a contention intensity (CI) value and invokes a centralized scheduler only when a predetermined threshold is reached. In its simplest version, a conflicting transaction is inserted into a global queue and the scheduler assures that only one transaction is executing at a time. This approach allows for nearly zero overhead when enough parallelism exists, and appropriately serializes execution when the conflict rate rises. ATS allows for a very efficient implementation (only a global lock), which makes the solution very attractive. However, serializing might not be the best solution in certain cases. Our `CILUTS` heuristic shows that executing another transaction in place of the doomed one might yield better results.

CAR-STM [5] keeps a transaction queue and a thread (TQ thread) for each available core in the system. When a transaction is about to start, the dispatcher selects which queue to insert the transaction and passes the control to the corresponding TQ thread. The general idea is to insert transactions that are likely to conflict in the same queue beforehand,

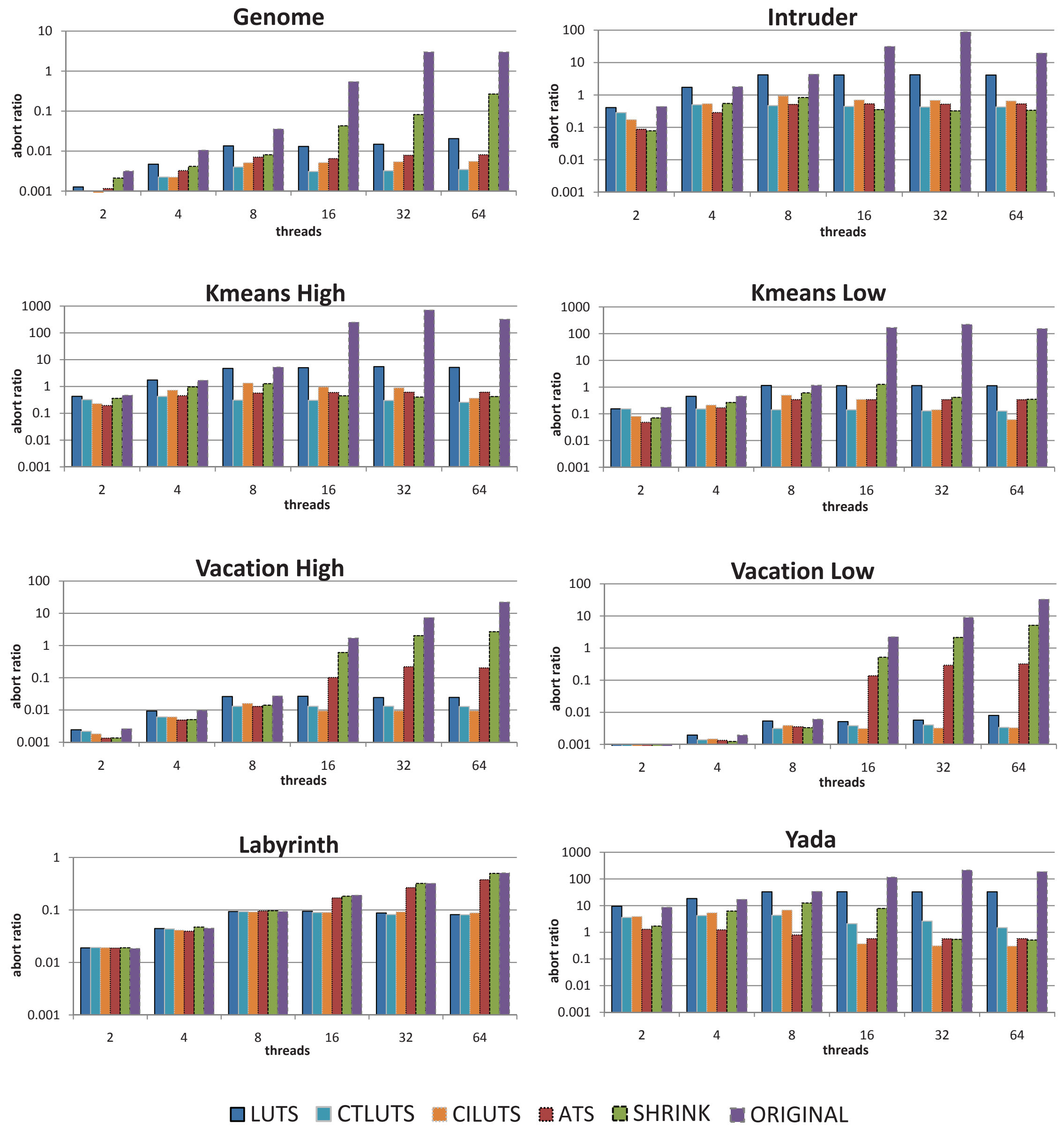


Figure 5: Abort ratio on STAMP benchmark.

thus avoiding collisions. At runtime, CAR-STM can reduce transaction conflicts by moving an aborted transaction to the TQ of the conflicting transaction, therefore serializing their execution. Ansari et al. [1] present a similar runtime technique called steal-on-abort. As with CAR-STM, the abortee transaction is moved to the thread queue of the aborter so as to avoid further conflicts. Differently from CAR-STM, the authors investigate different strategies when inserting the aborted transaction into the other thread queue (ie., insert at the tail or head). Our work bears some resemblance with CAR-STM in the sense that a fixed number of dedicated threads are assigned to execute transactions. CAR-STM implementation, however, relies on system-level locks and condition variables to synchronize transaction threads with traditional threads. LUTS, by design, maintains only transaction threads, thus avoiding the additional synchronization overhead.

Shrink [7] is a scheduling technique that uses conflict prediction in order to avoid a transaction that is likely to abort from starting. The prediction is based on access patterns of past transactions of a given thread. More specifically, it keeps a signature (a bloom filter) for each of the read and write sets in a per thread basis. The authors suggest using the temporal locality principle to maintain the read set signature, whereas the writes of an aborting transaction are employed as the prediction strategy for the write set signature. Shrink is only activated when a certain contention threshold is met, therefore avoiding a large overhead in low concurrency levels. Our results show that albeit low, the technique still might exhibit some overhead. This overhead can be reduced by choosing an optimal configuration, but doing so on a per-application basis has its drawbacks.

Blake et al. [2] present a proactive scheduling mechanism that also uses prediction to avoid starting doomed transactions. Their system stores a confidence value for each pair of transactions that represent the likelihood of a conflict occurring in the future between the pair. The scheduler relies on the confidence value to decide whether a transaction should start, wait, or give opportunity to another transaction. Maldonado et al. [13] investigate performance issues when implementing serialization at the user level and offer kernel-level implementations to reduce synchronization overhead. The authors present and compare different strategies both for user and kernel levels. They also describe a technique that reduces the thread priority for scenarios where non-determinism may occur, and a time-slice extension mechanism in order to reduce conflicts of long-running transactions. LUTS also aims at reducing the synchronization overhead, but does not require changing the operating system kernel. Proactive heuristics can be easily created using LUTS scheduling capabilities which could further increase performance. A key distinction from prior works is that LUTS makes it possible to choose another transaction from the scheduler queue instead of serializing the execution.

6 Conclusions

We have introduced in this paper LUTS: a Lightweight User-level Transactional Scheduler. LUTS effectively controls the contention level in pseudo parallelism scenarios, and provides the means to increase system performance by avoiding starting transactions that are likely to abort in the near future. In order to accomplish its goal, LUTS relies on two key factors:

(i) the number of spawned threads is limited to the number of available processor cores, and (ii) STM libraries can exert influence on the scheduling policy and devise new proactive conflict-avoidance heuristics. LUTS scheduling capabilities is in sharp contrast with prior works, which primarily resorted to serialization.

We have presented LUTS design, a prototype implementation and two proactive conflict-avoidance heuristics, `CILUTS` and `CTLUTS`, developed using LUTS scheduling capabilities. Our evaluation show that LUTS has better performance results than existing techniques in every STAMP benchmark program, reaching speedups up to 23.9x.

References

[1] Mohammad Ansari, Mikel Lujan, Christos Kotselidis, Kim Jarvis, Chris Kirkham, and Ian Watson. Steal-on-Abort: Improving transactional memory performance through dynamic transaction reordering. In *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, pages 4–18, January 2009.

[2] Geoffrey Blake, Ronald G. Dreslinski, and Trevor Mudge. Proactive transaction scheduling for contention management. In *Proceedings of the 42nd ACM/IEEE International Symposium on Microarchitecture*, pages 156–167, December 2009.

[3] Frederica Darema. The SPMD model: Past, present and future. In *Proceedings of the 8th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, page 1, 2001.

[4] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *20th International Symposium on Distributed Computing*, pages 194–208, September 2006.

[5] Shlomi Dolev, Danny Hendler, and Adi Suissa. CAR-STM: Scheduling-based collision avoidance and resolution for software transactional memory. In *Proceedings of the 27th Annual Symposium on Principles of Distributed Computing*, pages 125–134, August 2008.

[6] Aleksandar Dragojevic, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 155–165, June 2009.

[7] Aleksandar Dragojevic, Rachid Guerraoui, Anmol V. Singh, and Vasu Singh. Preventing versus curing: Avoiding conflicts in transactional memories. In *Proceedings of the 28th Annual Symposium on Principles of Distributed Computing*, pages 7–16, August 2009.

[8] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th Symposium on Principles and Practice of Parallel Programming*, pages 237–246, February 2008.

[9] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Polymorphic contention management. In *19th International Symposium on Distributed Computing*, pages 303–323, September 2005.

[10] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a theory of transactional contention managers. In *Proceedings of the 24th Annual Symposium on Principles of Distributed Computing*, pages 258–264, July 2005.

[11] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory.* Morgan & Claypool Publishers, 2nd edition, June 2010.

[12] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, pages 92–101, July 2003.

[13] Walther Maldonado, Patrick Marlier, Pascal Felber, Adi Suissa, Danny Hendler, Alexandra Fedorova, Julia L. Lawall, and Gilles Muller. Scheduling support for transactional memory contention management. In *Proceedings of the 15th Symposium on Principles and Practice of Parallel Programming*, pages 79–90, January 2010.

[14] Austen McDonald, JaeWoong Chung, Brian D. Carlstrom, Chi Cao Minh, H. Chafi, Christos Kozyrakis, and Kunle Olukotun. Architectural semantics for practical transactional memory. In *Proceedings of the 33rd International Symposium on Computer Architecture*, pages 53–65, June 2006.

[15] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[16] 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*, pages 267–275, 1996.

[17] Chi Cao Minh, JaeWoong Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 35–46, September 2008.

[18] Daniel Nicacio and Guido Araujo. Reducing false aborts in stm systems. In *Algorithms and Architectures for Parallel Processing*, volume 6081 of *Lecture Notes in Computer Science*, pages 499–510. 2010.

[19] William N. Scherer and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the 24th Annual Symposium on Principles of Distributed Computing*, pages 240–248, July 2005.

[20] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. A comprehensive strategy for contention management in software transactional memory. In *Proceedings of the 14th Symposium on Principles and Practice of Parallel Programming*, pages 141–150, February 2009.

[21] Haris Volos, Andres Jaan Tack, Neelam Goyal, Michael M. Swift, and Adam Welc. xCalls: safe I/O in memory transactions. In *Proceedings of the 4th European Conference on Computer Systems*, pages 247–260, April 2009.

[22] Richard M. Yoo and Hsien-Hsin S. Lee. Adaptive transaction scheduling for transactional memory systems. In *Proceedings of the 20th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, June 2008.