Noname manuscript No.
(will be inserted by the editor)

Privatizing Transactions for Lee's Algorithm in Commercial Hardware Transactional Memory

Ricardo Quislant and Eladio Gutierrez and Emilio L. Zapata and Oscar Plata

Received: date / Accepted: date

Abstract Lee's algorithm solves the path-connection problems that arise in logical drawing, wiring diagramming or optimal route finding. Its parallel version has been widely used as a benchmark to test transactional memory systems. It exhibits transactions of large size and duration that stress these systems exposing their limitations. In fact, Lee's algorithm has been proved to perform similar to sequential in commercial hardware transactional memory systems due to persistent capacity overflows.

In this paper we propose a novel approach to Lee's algorithm in the context of commercial hardware transactional memory systems. We show how the majority of the computation of the largest transaction, i.e. grid privatization and path calculation, can be executed out of the boundaries of the transaction thus reducing the size requirements. We leverage the correctness criteria of lazy subscription fallback locks to ensure a correct execution. This novel approach uses transactional memory extensions from commercial processors from a different point of view, not needing either early release or open nested transaction features that are not yet implemented in these systems. We propose an application programming interface to facilitate the task of the programmer.

Experiments are carried out with the Intel Core and IBM Power8 architectures, showing speedups around 3.5x over both the standard transactional version of the algorithm and the sequential for certain grid inputs and 4 threads. We also compare our proposal with a software transactional memory LeeTM approach.

Keywords Hardware transactional memory · Lee's algorithm · Early release · Open transactions · Lazy subscription

Department of Computer Architecture
University of Málaga, Spain, 29071
E-mail: {quislant, eladio, zapata, oplata}@uma.es

1 Introduction

Hardware transactional memory (HTM) was first proposed by Herlihy and Moss [1] with the intention of making lock-free synchronization efficient and easy to use, and thus avoiding the problems associated with locks. After two decades of intensive research by the computer science and engineering community, chip manufacturers release HTM extensions along with their multiprocessors. This is the case of Intel Core [2] and IBM Power8 [3] architectures. Commercial HTM extensions provide a best-effort solution where hardware transactions are limited in size and duration by hardware constraints, and a software fallback path must be provided to ensure forward progress.

To test transactional memory (TM) proposals in general and commercial HTM systems in particular, the research community has often made use of the STAMP [4] benchmark suite. The suite includes a TM implementation of Lee's algorithm renamed as *Labyrinth*. Lee's algorithm [5] solves the path-connection problems that arise in logical drawing, wiring diagramming or optimal route finding. Unfortunately, such an implementation does not comply with the properties that should have a code to be a potential candidate for HTM [6], such as low contention and critical sections of moderate size. It exhibits transactions of large size and duration that stress HTM systems exposing their limitations. In fact, Labyrinth has been proved to yield sequential performance in commercial HTM systems due to persistent capacity overflows [7–9]. Several papers propose methods of controlling and scheduling transactions to reduce contention [10–13], but in this case the problem is related to transaction size.

In this paper we propose a novel approach to Lee's algorithm in the context of commercial HTM systems. The contributions are the following:

- We show how the majority of the computation of the largest transaction, i.e. the grid privatization and the path calculation, can be executed out of the boundaries of the transaction thus reducing the size requirements of the application.
- We leverage the correctness criteria of lazy subscription fallback locks to ensure a correct execution. Lazy subscription for HTM has been used in the implementation of improved global lock fallbacks [14] and consists in subscribing to the lock at the end of the transaction instead of at the beginning. In this way, concurrency between non-transactional fallback code and transactional code is allowed.
- Our approach uses transactional memory extensions from commercial processors from a different point of view, not needing either early release [15] or open nested [16] transaction features that are not yet implemented in these systems.
- We propose an application programming interface (API) that hides the added complexity from the user.

We have evaluated our proposal using Intel Core and IBM Power 8 architectures. Experimental results show speedups of around 3.5x over both the standard transactional version of the algorithm and the sequential for certain grid inputs and 4 threads. We also compare our proposal with a software transactional memory LeeTM approach obtaining better results.

The remainder of this work is organized as follows. Section 2 discuss Lee's algorithm, the parallel versions found in the bibliography and the concept of lazy sub-

```
1  void sequentialLee(gridType *grid, pairType* pairList, int bendCost) {
2    pairType pair;
3
4    while(pair = pickSrcDestPair(pairList)) { //While there is a pair
5      if(expansion(pair, grid))  //If destination reached
6        backtracking(pair, grid, bendCost);
7      resetExpansion(grid);
8    }
9  }
```

Fig. 1: Lee's sequential algorithm kernel code.

scription. Section 3 introduces our proposal for commercial HTM systems and outlines our API and its semantics. Section 4 provides an informal correctness proof for our algorithm. Experimental methodology is presented in Section 5. Finally, Section 6 discusses the results and Section 7 draws the conclusions.

2 Background and Related Work

2.1 Lee's Algorithm

An algorithm for path connections was first proposed by Lee in 1960 [5] to solve problems of logical drawing, wiring diagramming and optimal route finding. This algorithm is able to find the shortest path between two points, when possible, and in the presence of obstacles such as edges, other previously calculated paths, etc.

The code for Lee's sequential algorithm is shown in Figure 1. The input (line 1) consists of a list of source-destination coordinate pairs (`pairList`), a grid that can have three dimensions (`grid` and the cost of bending a path (`bendCost`). The list of coordinate pairs can be sorted by distance from longer to shorter if we want the longest paths to encounter as few obstacles as possible. The bending cost can be important for certain applications where we want a minimum number of bends, e.g. bends in printed circuit board tracks may cause attenuation of certain signal frequencies.

Lines 4 to 8 show the kernel code of the sequential Lee's algorithm with its four phases:

1. *Initial state*. One pair of source-destination coordinates is picked from the list of pairs in line 4. The pair is placed in the grid.
2. *Expansion*. We expand from source to destination labelling each grid cell with a weight which is the number of steps needed to reach that cell (line 5).
3. *Backtracking*. If the expansion reached the destination point (line 5) we backtrack from destination to source by choosing the lowest weight neighbour at each cell (line 6). The bending cost is added to the weight of a cell if taking that cell implies changing the direction of the path. If even after adding the bending cost two neighbouring cells show the same weight, one of them is taken indistinctly.
4. *Reset expansion*. The expansion is reset (line 7) by eliminating the weights from the grid. The calculated path is left marked as an obstacle.

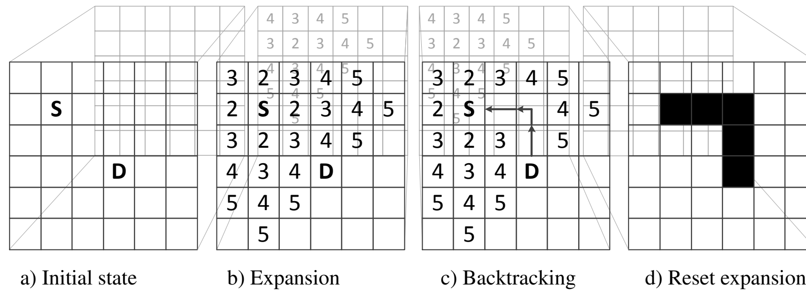

Fig. 2: Lee's algorithm phases in a three dimensional grid of $6 \times 6 \times 2$. S: Source, D: Destination.

Figure 2 shows an example with the four phases of Lee's algorithm using a $6 \times 6 \times 2$ grid. A source(S)-destination(D) coordinate pair is placed in the grid to set up the initial state, Figure 2a.

Figure 2b shows the expansion from S to D. Each neighbour of S is labelled with a weight by adding 1 to the weight of S, which is initialized to 1. The neighbours are searched through the three dimensions of the grid. Notice that this algorithm follows the Manhattan geometry. The expansion ends when D is reached, with the weight of 5 in this particular case. A path cannot be traced if an expansion iteration cannot label any neighbour and D has not been reached.

Since D was reached, the algorithm goes to the backtracking phase, Figure 2c. Starting from D we search for those neighbours with the lowest weight, west and north in this case. The path has not been started yet so a bending cost does not apply. Then, one of the neighbours is chosen randomly, which is the one in the north in this example. Now, moving to the new cell we find two neighbours with the lowest weight of 3. However, the one in the west has to be added the bending cost giving as a result a movement to the north. In this new cell there is one lowest weight neighbour to the west. It has to be added the bending cost. Notice that if the bending cost is greater than one the path will not be traced as all neighbours will be greater than the weight of the cell (> 1 weights yield straight paths only).

Figure 2d shows the final phase where the labelled cells are cleared except for those that comprise the calculated path.

2.2 Transactional Memory Approaches to Lee's Algorithm

Parallel versions of Lee's algorithm are difficult to find in the bibliography. Although it seems an inherently parallel algorithm where the calculation of each path is the unit of parallelism (each iteration of the while loop in Figure 1), traditional synchronization methods can lead to different problems. If we protect each loop iteration (lines 5–7) with a coarse grain lock the execution is serialized. Conversely, protecting each grid cell with a fine grain lock may lead to deadlock scenarios. Actually, Won et al. [17] and Yen et al. [18] approach the parallelization of the algorithm from another point of view. They partition the grid among processors using different map-

```
1  void threadParallelLee(gridType *globalGrid, pairType *pairList, int bendCost) {
2    gridType *localGrid;
3    pathType *path;
4    while(TRUE) {   //While there is a pair
5      pairType *pair;
6      xbegin();
7        pair = pickSrcDstPair(pairList);
8      xend();
9      if(!pair) break;  //Terminate thread work
10
11     xbegin();                                       xbeginOpen();
12       gridCopyFromTo(globalGrid, localGrid); } <-> {   gridCopyFromTo(globalGrid, localGrid);
13       earlyRelease(globalGrid);                      xendOpen();
14       if(expansion(pair, localGrid)) {
15         path = backtracking(pair, localGrid, bendCost);
16         for(int i=1; i < length(path); i++) {  //Copy path to global grid
17           if(globalGrid[path[i]] != GRID_POINT_EMPTY) //Validation
18             xabort();
19           globalGrid[path[i]] = GRID_POINT_FULL;
20         }
21       }
22     xend();
23   }
24 }
```

Fig. 3: Watson et al. suggestions for an optimized parallel TM Lee's algorithm.

ping strategies. Watson et al. state so in [19], and propose three different parallel transactional-memory-based versions of Lee's algorithm.

First, Watson et al. simply enclose the three main phases of the algorithm (lines 5–7 in Figure 1) in a transaction that works with the global grid. Such an approach exhibits poor parallelism since the writes to the global grid performed by a transaction in the expansion phase abort other transactions whose expansion used one of the grid cells written.

A second approach by Watson et al. deals with the privatization of the grid. Each transaction first expands reading from the global grid and writing to a private one, and then performs the backtracking phase by reading from the private grid and writing to the global one. Resetting the expansion is no longer needed in the global grid but in the private. This version tries to avoid the aborts due to expansions in the global grid and performs slightly better than the first approach. However, if expansions read too many global grid cells there is a high probability of conflicts when writing back the calculated paths to the global grid.

The third approach in [19] optimizes the algorithm by holding only the locations of the calculated path in the transaction's data set. In this manner, the transaction first reads the entire global grid and writes it in its private grid. Subsequently, those reads are cleared from the transaction's read set. The result is a significant increase in performance. However, Watson et al. do not tackle the implementation using TM constructs, since they seek for an implementation independent evaluation. On the contrary, they discuss that the algorithm could use early release [15] or open nested transactions [16]. In fact, they provide a benchmark using early release in the Lee-TM benchmark suite [20]. The same approach is used in STAMP's Labyrinth.

Figure 3 shows the thread kernel code following the suggestions of Watson et al. Line 12 copies the global grid to the local with the protection of the transactional system, so that a concurrent write of a path from other transaction aborts the copy

of the grid to start from an updated one. Line 13 executes the early release of the global grid, thus deleting those reads from the read set of the transaction. Next, the expansion and the backtracking are performed with the local grid, and the resulting path is written to the global one in lines 16 to 20. We need to check whether *all the grid cells comprising the path are cleared* in the grid (line 17) and abort the transaction if not (line 18), as another transaction may have committed a path between our early release instruction and the copy of our path. Lines 12 and 13 can be replaced by the open transaction on the right-hand side of the figure with similar results.

Early release and open transactions are features not yet included in commercial HTM systems. Consequently, the versions of the algorithm provided by Lee-TM and STAMP that make use of the early release feature can be executed only in software TM (STM) systems such as DSTM2 [21] or TinySTM [22], as well as in HTM simulators. This matter makes the algorithm behave differently with HTM, merely serializing the execution [9], than with STM, where more speedup can be achieved [4]. In Section 3 we propose a method to implement the optimized parallel TM version of Lee's algorithm for commercial HTM systems without the need for the aforementioned features.

2.3 Lazy Subscription

A fallback path is needed in best-effort HTM systems to provide an alternative to transactional execution when transactions cannot progress (because of hardware constraints, persistent conflicts,etc.). Such a fallback usually comprises the same code of the transaction enclosed by a global lock. All transactions have to subscribe to the lock by reading it before performing any computation so that they are aborted when the fallback acquires the lock. Thus, transactional and non-transactional code cannot coexist. Insights into fallback paths can be found in [8,23].

Lazy subscription to the fallback lock is used in the implementation of improved global lock fallbacks [14] and consists in subscribing to the lock at the end of the transaction instead of at the beginning. This way, concurrency between non-transactional fallback code and transactional code is allowed. To ensure a correct execution, the HTM system must comply with strong isolation [24], by which non-transactional memory accesses can abort those transactions holding the accessed memory locations in their data sets, and sandboxing, which prevents the propagation of errors from inside a hardware transaction to other transactions. Thus, writes caused by reading inconsistent state from non-transactional fallback computation are not visible to other threads. Section 4 expands on the correctness of lazy subscription and how we use it in our proposal.

Figure 4 shows an API that implements a fallback path with lazy global lock subscription. The `xbegin()` and `xend()` functions are redefined to execute a fallback path after a number of aborts (`globalRetries` in lines 5 and 13). Line 8 shows a busy wait before executing the transaction to avoid the Lemming effect [25]. The lock subscription is performed by lines 14 and 15. In case of a conventional subscription, such lines would be between lines 9 and 10. Note that code in lines 4 to 8 are executed non-transactionally. Label `ret` defines the point where a transaction is resumed after

```
1  #define XBEGIN()                                    \
2  {                                                   \
3    int retries = 0;                                  \
4  ret: retries++;                                     \
5    if (retries > globalRetries) {                    \
6      while(!acquireLock(&globalLock));               \
7    } else {                                          \
8      while(globalLock); /*Wait if lock is taken*/    \
9      xbegin(ret);                                    \
10   }
11
12 #define XEND()                                      \
13   if (retries <= globalRetries) {                   \
14     if (globalLock) xabort();/*Lazy subscription*/\
15     xend();                                         \
16   } else {                                          \
17     releaseLock(&globalLock);                       \
18   }                                                 \
19 }
```

Fig. 4: Lazy global lock fallback API.

an abort. HTM extensions such as Intel's allow us to provide a label from which the execution resumes after an abort [26].

We are using a single global lock in the codes of the figure for the sake of simplicity, but a ticket lock is more convenient as stated in [23].

3 Lee's Algorithm Approach for Commercial HTM

In STM systems, each memory instruction in a transaction is explicitly instrumented to indicate that it has to be added to the data set of the transaction. Besides, STM systems can handle virtually infinite transaction sizes, and can provide the early release feature. Commercial HTM systems, on the contrary, are implicit systems where each memory access in a transaction is tracked by the underlying HTM mechanisms. Transactional data sets are constrained to the size of buffers or L1/L2 caches, and deleting memory locations from those sets is not allowed yet.

With these constraints, and even if we were provided with an early release instruction, HTM Lee's algorithm will perform similarly to the sequential algorithm whenever the grid set is larger than the cache/buffer. Hardware overflows would persistently cause transaction abort. This fact will happen in every other application whose transactional data set exceeds the hardware capacity.

Our purpose is to keep the transaction as short and small as possible. Therefore, in the case of Lee's algorithm, we propose to enclose only the copy of the calculated path to the global grid in the transaction (lines 16 to 19 in Figure 3), whereas the other phases of the algorithm are executed non-transactionally. Even the privatization of the grid. As in the optimized version of Watson et al., the transaction should first check if the path cells are cleared in the global grid or they have been written to by other transactions. If they have, the transaction must self-abort. We can do this in commercial systems by leveraging the fallback path handler and we propose an API to facilitate the task to the programmer and to provide a level of abstraction.

3.1 Privatizing hardware transaction API

Figure 5 shows our proposal for the API of privatizing hardware transactions. Four statements define the privatizing transaction and they divide it in three sections whose semantics are the following:

1. *Privatizing and execution section*: Delimited by macros `XBEGIN_PRIV()` (line 1) and `XVALIDATE_PRIV()` (line 9). In this section the user is meant to perform only reads to global memory. The programmer should privatize the data required by the algorithm and perform the execution phase of it. This section is executed non-transactionally.
2. *Validation section*: The programmer must validate the global reads performed in the previous section to ensure they have not changed. The code in this section is executed transactionally, and is enclosed by macros `XVALIDATE_PRIV()` and `XUPDATE_PRIV(isvalid)` (line 24). The result of the validation, whether true or false, has to be passed to the `XUPDATE_PRIV(isvalid)` statement as an argument. This section is not executed if the fallback is taken (line 10 shows the if clause whose scope is closed in line 26).
3. *Update section*: The code to commit the results obtained in the privatizing and execution section must be placed between macros `XUPDATE_PRIV(isvalid)` and `XEND_PRIV()` (line 28). The update section is protected by the same transaction of the validation section, and it is executed only if the validation was successful. If not, the transaction is aborted (line 25).

It should be noted that the API differentiates between an abort due to validation error and other aborts. If the abort cause was a validation error (line 14) or the transaction reached the retry limit, it is executed from the beginning (line 17). With our API, the user has not to deal with explicit aborts. Furthermore, we provide a lazy subscription fallback policy that encourage parallelism.

Finally, the fallback API shown in Figure 4 can coexist with our privatizing hardware transaction API, so that the programmer can choose what to use.

3.2 Privatizing HTM Lee

The standard version of HTM Lee's algorithm is basically the code in Figure 3 without either the early release or the open transactions features, which are not provided with commercial HTM systems. We must use the API in Figure 4 to include a fallback path. As we will see in Section 6, this approach serializes the execution.

Figure 6 shows our optimized HTM proposal of Lee's algorithm using our API for privatizing hardware transactions. In fact, both APIs are used. The standard of Figure 4 to get the pair from the global list of pairs (lines 6 to 8) and the privatizing API of Figure 5 for the main transaction (lines 11 to 28).

The main transaction starts by resetting the variable `path` in line 12. This variable is used in other sections of the transaction to know if the path was found, other than to store the path. It has to be initialized at the beginning as the first section of the privatizing transaction is non-transactional and writes are not undone on abort. Next, the

```
1  #define XBEGIN_PRIV()                              \
2  {                                                  \
3    int retries = 0, fromBegin = 1;                  \
4  begin:                                             \
5    if (retries > globalRetries) {                   \
6      while(!acquireLock(&globalLock)) ;             \
7    }
8
9  #define XVALIDATE_PRIV()                           \
10   if (retries <= globalRetries ) {                 \
11 validate:                                          \
12     if(!fromBegin) {                               \
13       retries++;                                   \
14       if(VALIDATION_ERROR == xabortCause()         \
15         || (retries > globalRetries)) {            \
16         fromBegin = 1;                             \
17         goto begin;                                \
18       }                                            \
19     }                                              \
20     fromBegin = 0;                                 \
21     while (globalLock); /*Avoid Lemming*/          \
22     xbegin(validate);
23
24 #define XUPDATE_PRIV(isValid)                      \
25     if (!isValid) xabort(VALIDATION_ERROR);        \
26   }
27
28 #define XEND_PRIV()                                \
29   if (retries <= globalRetries) {                  \
30     if (globalLock) xabort();/*Lazy subscription*/\
31     xend();                                        \
32   } else {                                         \
33    releaseLock(&globalLock);                       \
34   }                                                \
35 }
```

Fig. 5: Privatizing hardware transaction API.

bulk of the work is performed, namely, the grid privatization (line 13), the expansion (line 14) and the backtracking (line 15). If the path was successfully traced, the `path` variable will not be null and will hold a pointer to a vector of grid cell indices.

Whereas in Figure 3 validation and path copy is performed by the same loop, with our privatizing transaction API those steps are performed separately. Lines 16 to 24 perform the validation, where we define the variable `isValid` in line 17. If every point in the path is empty in the global grid then the path is valid. The result is passed as a parameter to `UPDATE_PRIV()`, which is in charge of explicitly aborting the transaction or not. Eventually, lines 24 to 28 update the global grid with the path.

The privatizing HTM Lee's algorithm keeps as simple as the standard algorithm by using our privatizing transaction API, and it yields better performance as we will see in Section 6.

4 Correctness

The most sensitive part of our proposal is the privatization of the global grid (line 13 in Figure 6), which is performed non-transactionally. The phases of expansion

```
1  void threadOptParallelLee(gridType *globalGrid, pairType *pairList, int bendCost) {
2    gridType *localGrid;
3    pathType *path;
4    while(TRUE) {  //While there is a pair
5      pairType* pair;
6      XBEGIN();
7        pair = pickSrcDstPair(pairList);
8      XEND();
9      if(!pair) break;  //Terminate thread work
10
11     XBEGIN_PRIV();
12       path = NULL; //Reset variables
13       gridCopyFromTo(globalGrid, localGrid);  //globalGrid read only
14       if(expansion(pair, localGrid))
15         path = backtracking(pair, localGrid, bendCost);
16     VALIDATE_PRIV();
17       bool isValid = TRUE;
18       if(path)
19         for(int i=1; i < length(path); i++)
20           if(globalGrid[path[i]] != GRID_POINT_EMPTY) {
21             isValid = FALSE;  //One path cell is not empty
22             break;            //Terminate validation
23           }
24     UPDATE_PRIV(isValid);
25       if(path)
26         for(int i=1; i < length(path); i++)
27           globalGrid[path[i]] = GRID_POINT_FULL;
28     XEND_PRIV();
29   }
30 }
```

Fig. 6: Privatizing HTM Lee's algorithm using our API.

and backtracking work with local variables and do not pose any problem in terms of correctness.

In the version of Watson et al. (Figure 3) such a privatization is protected by the transaction, and afterwards, the reads are erased from the read set with the early release feature. In a standard HTM version, where a normal transaction or the API in Figure 4 is used to enclose the Lee kernel, the access to the global grid would be also protected by the transaction. However, as the early release is not available, the result is a complete serialization. Next, we show how using the same correctness criteria of lazy subscription the privatization can be performed non-transactionally and still obtain a correct algorithm without the penalization of holding the entire global grid in the transaction's data set.

Figure 7 shows the different interleavings among transactions (Cases 1 to 3), and between transactions and fallbacks (Cases 4 to 8) that can arise in a parallel HTM system running our privatizing transactions. In Case 1 the transaction in the second thread (`Th2`) ends before the transaction in `Th1` privatizes the grid, so `Th1` finds an updated grid an no conflicts. Case 2 is less obvious. The privatization of the grid by `Th1` and the copy of the path by `Th2` happen at the same time. Then, a non-transactional read instruction from `Th1` aborts `Th2`'s transaction by means of strong isolation, thus keeping the grid of `Th1` updated. Finally, in Case 3, `Th2` commits its path to the global grid after `Th1` privatizes the grid. So `Th1` has calculated its path on a stale grid. Therefore, the path copy procedure has to read the path cells from

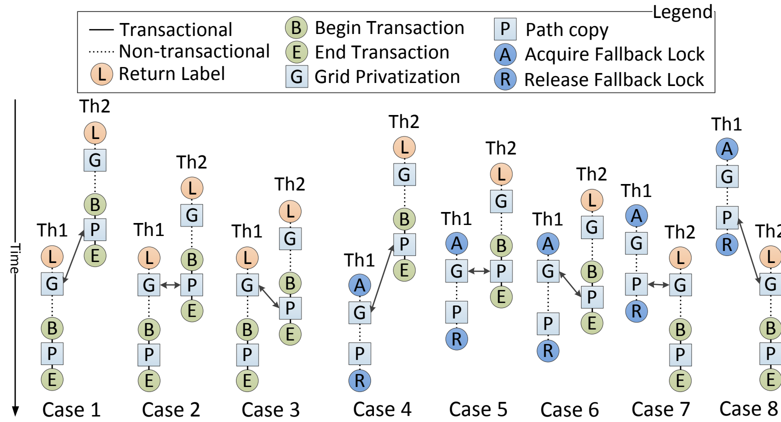

Fig. 7: Transaction-transaction and fallback-transaction interleavings for the privatizing HTM Lee.

the global grid to check if they are cleared (validation), as suggested by Watson et al. (see Section 2.2). If there is and obstacle the transaction is aborted by the API.

Regarding the interaction among fallbacks and transactions, Cases 4 and 8 show the simple cases where the fallback copies the grid after the end of the transaction and vice versa. Case 5, 6 and 7 are more complex. In Case 5 Th2 tries to end the transaction before the fallback ends. Th1 privatization may abort Th2's transaction due to strong atomicity. In Case 6, the copy of the path in Th2 happens between Th1's privatization and path copy. The lazy subscription to the fallback lock aborts Th2's transaction. It should be noticed that the fallback path cannot be aborted and its copy path procedure will never find a grid cell obstacle in its path. Lastly, in Case 7, Th2's transaction can partially read the path written by Th1's fallback. Then, Th1 will work with an stale grid as in Case 3. Reading the cells of the global grid, the validation section, solves the issue.

Lazy subscription may pose the problem of unsafe execution. This may happen when an inconsistent read from a transaction in parallel with non-transactional fallback code makes the transaction jump to a commit instruction, without performing the lock subscription. However, our algorithm does not perform jumps to `xend()` and, consequently, the lazy subscription in line 30 of Figure 5 is always executed.

5 Evaluation Methodology

For the evaluation of our proposal we use two different architectures that provide HTM extensions, Intel Core and IBM Power 8. Specifically, we have used the following commercial machines:

- Intel Core i5-4570 (Haswell microarchitecture): 4-core 3.2-GHz without simultaneous multi-threading (SMT) and 8 GB main memory.
- IBM PowerNV 8335-GCA: 2 by 10-core 3.5-GHz with 8 SMT threads and 512 GB main memory.

Table 1: Characteristics of the grids. Aborts and TCR(%) for Haswell (Hw) and Power8 (P8), 4 threads.

Dimensions (x,y,z)	Paths (n)	Max/Avg/Dev Path Length	Aborts / TCR(%) Hw Priv	Hw Std	P8 Priv	P8 Std
(128,128,3)	1024	7 / 4.1 / 1.2	8.8 / 99	6019.4 / 14	4.2 / 99	5359.5 / 16
(128,128,3)	512	7 / 4.1 / 1.2	4.8 / 99	2936.4 / 14	2.3 / 99	2657.6 / 16
(64,64,3)	64	94 / 42.9 / 22.1	35.9 / 78	358.5 / 16	229.1 / 30	346.0 / 17
(64,64,3)	48	91 / 39.8 / 21.6	16.8 / 85	274.1 / 16	169.2 / 31	259.8 / 17
(48,48,3)	64	71 / 33.5 / 16.6	21.5 / 86	360.2 / 16	15.5 / 89	342.1 / 17
(48,48,3)	48	69 / 31.1 / 16.1	11.8 / 89	275.2 / 16	8.0 / 92	258.4 / 17
(32,32,3)	96	59 / 30.5 / 13.0	34.3 / 85	540.8 / 16	71.1 / 73	512.6 / 16
(32,32,3)	64	58 / 27.9 / 12.1	23.2 / 85	365.6 / 16	26.6 / 83	341.5 / 17

Regarding HTM characteristics, the Intel Core uses the L1 cache for conflict detection and store buffering. The write set capacity for a transaction is within the size of the L1 cache (32 KB), but the read set is expanded to the order of megabytes with other resources [9]. On the other hand, the Power8 cores maintain a content addressable memory to keep track of transactional data sets with a total transaction capacity of 8 KB, both for the read set and the write set [9].

We evaluate the grids shown in Table 1. The two first larger grids were created randomly specifying a maximum length for the path, shorter than in the other grids to favour the small write set capacity of the HTM systems used for evaluation. The rest are taken from the inputs released with the Labyrinth benchmark from the STAMP suite [4]. The maximum, average and standard deviation for path lengths are taken from the sequential application. The algorithms evaluated are those in Figures 3 (the standard Lee with neither early release nor open transactions) and 6 with the `globalRetries` parameter set to 5. We also use a ticket lock instead of the single global lock as stated in [23] and a loop to avoid the Lemming effect [25] as shown in the API's in Figures 4 and 5.

In Section 6.3 we compare our HTM proposals with a STM implementation of Lee's algorithm. Details on the methodology followed for the comparison are described in the section.

6 Results

Figure 8 shows the speedup of the privatizing and the standard HTM Lee's algorithms on the Intel Core Haswell and the IBM Power8 machines. The speedup is over the sequential application executed in the corresponding machine. We spawn up to 8 threads so that each transaction can be executed in an exclusive physical core. The results for the Intel processor are limited to 4 cores. Figure 9 shows a breakdown of the abort reason.

6.1 Intel Core

On the Intel Core machine, the grids with dimensions (32,32,3) and (48,48,3) yields around 2× speedup over the sequential and standard versions of the algorithm with

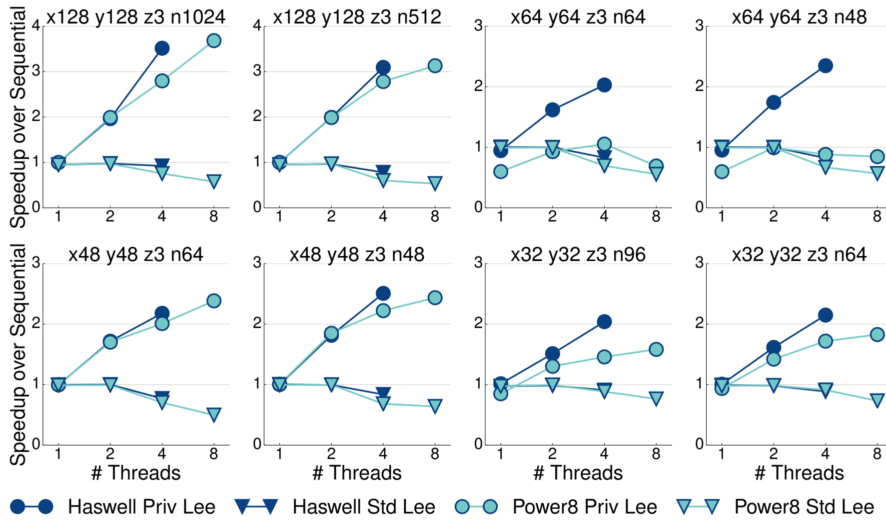

Fig. 8: Speedup of the privatizing and standard HTM algorithms with Intel Haswell and Power8.

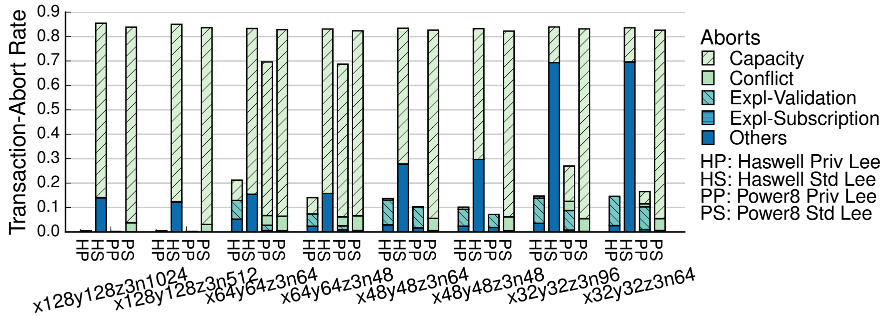

Fig. 9: Transaction-abort rate and abort breakdown with 4 threads.

4 cores. Table 1 shows a significant reduction on aborts with respect to the standard parallel version. The standard version copies the grid inside the transaction, and hardware resources are insufficient to keep track of the entire grid. Consequently, transactions are aborted when privatizing the grid until they meet the number of retries. Then, the execution is serialized and the performance falls to that of the sequential algorithm or lower. Other than the two transactions shown in Figure 6 there is another transaction for each thread to insert the calculated path in another global list but it rarely takes the fallback path and is low contended.

Our privatizing parallel HTM approach reduces the transaction footprint to the length of the path so the hardware capacity of Intel Core can accommodate the majority of the transactions.

The grids with dimensions (64,64,3), exhibit larger paths in average and maximum. Therefore the transactions in our privatizing HTM Lee are larger as well and

there are more capacity aborts (see Figure 9). Table 1 also shows the TCR percentage, transaction commit ratio, calculated as $\frac{commits}{commits+aborts} \times 100$. For the (64,64,3) grid and 64 paths the TCR decreases under 80%. In spite of these values the performance still is above 2 times the sequential with 4 threads as the number of aborts is not high.

The larger grids with dimensions (128,128,3) have shorter paths that allow our privatizing HTM approach to scale up to 3.5× over sequential and even more over the standard version with 4 threads. The TCR is almost 100% compared to a very low 14% of the standard parallel version which aborts every single kernel transaction due to capacity overflow (see Figure 9). The few aborts of our proposal are due to conflicts between paths (the explicit abort when validating the path to the grid — Expl-Validation in Figure 9), capacity aborts because of insufficient cache set associativity or capacity and other aborts (interrupts, descheduling, page faults, etc.).

The Lemming loop could be elided in the Haswell architecture as the hardware communicates that a transaction may succeed on a retry through a special value of the abort status word. This mechanism is not available in the Power8 architecture so we obviate it and use the Lemming effect avoidance loop instead (lines 8 and 21 in Figures 4 and 5 respectively). This justifies the increase in other aborts for Haswell in Figure 9.

6.2 IBM Power8

The IBM Power8 machine behaves differently for the smaller grids with longer paths. The 8 KB transactional buffer (see Section 5) limits the maximum size of a transaction compared with the greater size of the Intel Core architecture, regardless of the full associativity of the Power8 buffer. Thus, for the grid with dimensions (32,32,3) Power8 yields worse performance than Intel Haswell. As shown in Table 1, the average path length is similar to that of the (48,48,3) grid, but the standard deviation is lower, so the path lengths are closer to the average, and consequently the transactions are larger. Our privatizing HTM approach suffers from more capacity aborts in Power8 for that reason (see Figure 9), and the TCR is lower with Power8 than with Haswell.

The results for the grid with dimensions (48,48,3) are similar to those of Intel because the paths are more disperse from the average and there are more smaller transactions that fit in the buffer. However, the performance is worse for the (64,64,3) grid when using our privatizing HTM version of the algorithm due to the longer paths and consequently larger transactions. We find more than half the TCR with Power8 than with Intel Haswell, mainly due to the kernel transaction.

In fact, the privatizing parallel version performs worse than the sequential and the standard version with one thread. The reason is that the privatizing HTM version does more work (privatization, expansion, backtracking) before beginning the transaction to copy the path to the grid. If it is eventually aborted because of capacity overflow the transaction restarts from the validation point with a high probability of being aborted again, until the fallback is taken. The standard parallel version of the algorithm, though, is aborted earlier in the privatization phase and the penalization is negligible.

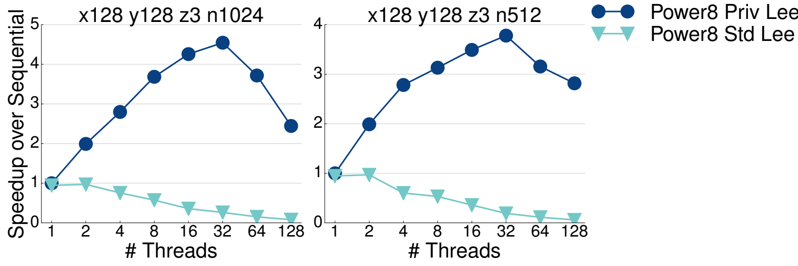

Fig. 10: Speedup of the privatizing and standard HTM algorithms with Power8 up to 128 threads.

This problem might be alleviated by identifying capacity aborts so that the transaction can take the fallback path immediately after them. Furthermore, we can implement different counters with different limits depending on the abort event as proposed in [9].

Finally, for the grids with dimensions (128,128,3) and short paths the results bear a resemblance to those obtained with the Intel Core machine. As the paths are short, so are the transactions, that fit perfectly in the Power8 buffer. In order to see to what extent the privatizing HTM algorithm scales when having a well conditioned input, Figure 10 shows the results with up to 128 threads. The speedup increases as the system is able to map each thread to a physical core, reaching about 4.5× as much performance as the sequential. From 16 threads on, the speedup decreases due to multiple threads in the same core sharing the transactional resources and provoking more capacity aborts.

6.3 Comparison with STM LeeTM

LeeTM [20] provides two real-world printed circuit board (PCB) input files with the benchmark, which are depicted in Figure 11. We have used them to compare our HTM proposal with STM LeeTM. However, we have to take into account several differences before interpreting the results shown in the figure:

- *Different algorithms:* LeeTM implements an algorithm focused on wiring diagramming where there are always two layers to deploy the wiring, e.g. the front and the back of a PCB. Consequently, the algorithm has a freedom degree on the z coordinate of the source and destination cells. Our proposal is implemented starting from the STAMPS's Labyrinth code which is not focused on wiring only. It traces paths from point to point whose coordinates are all defined.
- *Different input formats:* Due to the freedom degree of LeeTM, the PCB input files only define the x and y coordinates for the source and destination cells. To convert this format to the (x,y,z) format used by our algorithm we have assigned a z value for each (x,y) such that $z_{Destination} = z_{Source} = 0$ if $abs(x_{Destination} - x_{Source}) > abs(y_{Destination} - y_{Source})$, and $z_{Destination} = z_{Source} = 1$ otherwise. LeeTM does such assignation but it can change dynamically if the cell is not empty.

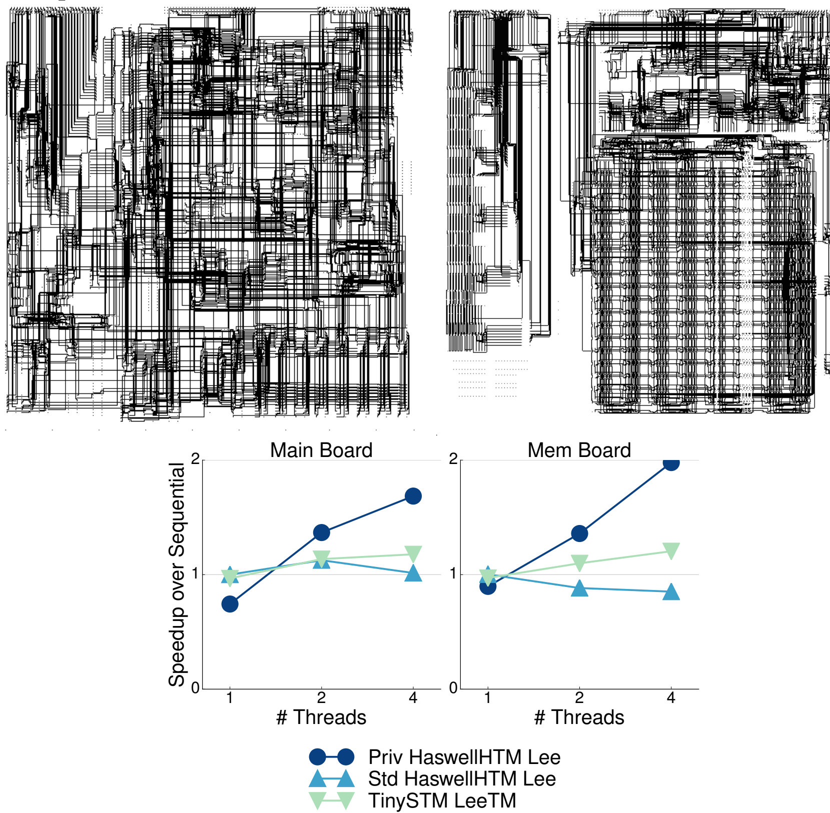

Fig. 11: Speedup of the privatizing and standard HTM algorithms along with the LeeTM algorithm using TinySTM in the Haswell machine.

– *Different TM systems:* Our proposed Lee algorithm is designed for commercial HTM systems whereas LeeTM is designed for STM. We have used the version for TinySTM of LeeTM provided in [27].

Given the differences above, the resulting wiring diagrams will turn out differently depending on the algorithm used. Furthermore, the LeeTM algorithm is implemented in C++ and with the STM system the overall execution time is much greater than our C HTM algorithm. Figure 11 shows near twofold speedup over HTM sequential for our privatizing HTM Lee. The standard HTM Lee is still similar or worse than the sequential. On the other hand, the TinySTM LeeTM version speedup shows a slight benefit over the LeeTM sequential version, although it settles quite below our HTM proposal.

7 Conclusions

In this paper we propose a new approach to Lee's algorithm for commercial HTM systems. The current proposals need features of a TM system such as early release or open transactions that are not yet provided with commercial HTM extensions.

We propose an implementation of Lee's algorithm without using these features by extending the correctness criteria of lazy subscription and leveraging the fallback path handler. The main phases of the algorithm are executed non-transactionally so the transaction size is reduced, thus being more amenable to be managed by the limited resources of HTM systems. We propose an API for our so-called privatizing transactions that facilitates the task of the programmer.

The evaluation with Intel Core and IBM Power8 architectures shows around $3.5\times$ speedup over the sequential and parallel applications with certain grid inputs and 4 threads. A twofold speedup is gained with real-world printed circuit board inputs comparing with the lower performance of a software TM system.

Our proposal might be used to help other codes showing a similar structure to Lee's algorithm, with a privatization and a final update of a global structure.

Acknowledgements

This work has been supported by the Government of Spain under project TIN2013-42253-P and TIN2016-80920-R, and Junta de Andalucía under project P12-TIC-1470.

References

1. Herlihy, M., Moss, J.: Transactional memory: Architectural support for lock-free data structures. In: 20th Ann. Int'l. Symp. on Computer Architecture (ISCA'93), pp. 289–300 (1993)
2. Yoo, R.M., Hughes, C.J., Lai, K., Rajwar, R.: Performance Evaluation of Intel Transactional Synchronization Extensions for High-performance Computing. In: Int'l Conf. on High Performance Computing, Networking, Storage and Analysis (SC'13), pp. 19:1–19:11 (2013)
3. Cain, H.W., Michael, M.M., Frey, B., May, C., Williams, D., Le, H.: Robust Architectural Support for Transactional Memory in the Power Architecture. In: 40th Ann. Int'l. Symp. on Computer Architecture (ISCA'13), pp. 225–236 (2013)
4. Minh, C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford Transactional Applications for Multi-Processing. In: IEEE Int'l Symp. on Workload Characterization (IISWC'08), pp. 35–46 (2008)
5. Lee, C.Y.: An Algorithm for Path Connections and Its Applications. IRE Transactions on Electronic Computers **EC-10**(3), 346–365 (1961)
6. Schindewolf, M., Bihari, B., Gyllenhaal, J., Schulz, M., Wang, A., Karl, W.: What Scientific Applications Can Benefit from Hardware Transactional Memory? In: Int'l. Conf. on High Performance Computing, Networking, Storage and Analysis (SC'12), pp. 90:1–90:11 (2012)
7. Goel, B., Titos-Gil, R., Negi, A., Mckee, S.A., Stenstrom, P.: Performance and Energy Analysis of the Restricted Transactional Memory Implementation on Haswell. In: 28th Int'l Symp. on Parallel and Distributed Processing (IPDPS'14), pp. 615–624 (2014)
8. Machado Pereira, M., Gaudet, M., Nelson Amaral, J., Araujo, G.: Study of hardware transactional memory characteristics and serialization policies on Haswell. Parallel Computing **54**, 46–58 (2016)
9. Nakaike, T., Odaira, R., Gaudet, M., Michael, M.M., Tomari, H.: Quantitative comparison of hardware transactional memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8. In: 42nd Ann. Int'l. Symp. on Computer Architecture (ISCA'15), pp. 144–157 (2015)

10. Ansari, M.: Weighted adaptive concurrency control for software transactional memory. The Journal of Supercomputing **68**(3), 1027–1047 (2014)
11. Atoofian, E.: Improving performance of software transactional memory through contention locality. The Journal of Supercomputing **64**(2), 527–547 (2013)
12. Chen, C.J., Chang, R.G.: A priority scheduling for tm pathologies. The Journal of Supercomputing **71**(3), 1095–1115 (2015)
13. Harvey, H.H., Mao, Y., Hou, Y., Sheng, B.: EDOS: Edge assisted offloading system for mobile devices. In: Int'l. Conf. on Computer Communication and Networks (ICCCN'17), pp. 1–9 (2017)
14. Calciu, I., Shpeisman, T., Pokam, G., Herlihy, M.: Improved Single Global Lock Fallback for Best-effort Hardware Transactional Memory. In: 9th Workshop on Transactional Computing (TRANSACT'14) (2014)
15. Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N.: Software transactional memory for dynamic-sized data structures. In: 22nd Ann. Symp. on Principles of distributed computing (PODC '03), pp. 92–101 (2003)
16. Moravan, M.J., Bobba, J., Moore, K.E., Yen, L., Hill, M.D., Liblit, B., Swift, M.M., Wood, D.A.: Supporting Nested Transactional Memory in logTM. In: 12th Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06), pp. 359–370 (2006)
17. Won, Y., Sahni, S.: Maze routing on a hypercube multicomputer. The Journal of Supercomputing **2**(1), 55–79 (1988)
18. Yen, I.L., Dubash, R.M., Bastani, F.B.: Strategies for mapping Lee's maze routing algorithm onto parallel architectures. In: Int'l. Parallel Processing Symposium, pp. 672–679 (1993)
19. Watson, I., Kirkham, C., Lujan, M.: A study of a transactional parallel routing algorithm. In: 16th Int'l. Conf. on Parallel Architecture and Compilation Techniques (PACT '07), pp. 388–398 (2007)
20. Ansari, M., Kotselidis, C., Watson, I., Kirkham, C., Luján, M., Jarvis, K.: Lee-TM: A Non-trivial Benchmark Suite for Transactional Memory, pp. 196–207 (2008)
21. Herlihy, M., Luchangco, V., Moir, M.: A Flexible Framework for Implementing Software Transactional Memory. In: Conf. on Object-oriented Programming Systems, Languages, and Applications (OOPSLA'06), pp. 253–262 (2006)
22. Felber, P., Fetzer, C., Riegel, T.: Dynamic Performance Tuning of Word-based Software Transactional Memory. In: Symp. on Principles and Practice of Parallel Programming (PPoPP'08), pp. 237–246 (2008)
23. Quislant, R., Gutierrez, E., Zapata, E.L., Plata, O.: Insights into the Fallback Path of Best-Effort Hardware Transactional Memory Systems. In: Int'l. Conf. on Parallel Processing (Euro-Par'16), pp. 251–263 (2016)
24. Martin, M.M.K., Blundell, C., Lewis, E.: Subtleties of Transactional Memory Atomicity Semantics. IEEE Computer Architecture Letters **5**(2), 17 (2006)
25. Dice, D., Herlihy, M., Lea, D., Lev, Y., Luchangco, V., Mesard, W., Moir, M., Moore, K., Nussbaum, D.: Applications of the Adaptive Transactional Memory Test Platform. In: 3rd Workshop on Transactional Computing (TRANSACT'08) (2008)
26. Intel: Intel(R) Architecture Instruction Set Extensions Programming Reference. Tech. Rep. February (2012)
27. Dragojevic, A.: TinySTM version of LeeTM. University of Manchester. http://apt.cs.manchester.ac.uk/projects/TM/LeeBenchmark/