Transaction Processing Core for Accelerating Software Transactional Memory

Ferad Zyulkyarov[12], Milos Milovanovic[12], Osman S. Unsal[1], Adrian Cristal[1], Eduard Ayguade[12], Mateo Valero[12], Tim Harris[3]

[1]Barcelona Supercomputing Center, Barcelona, Spain

[2]Universitat Politècnica de Catalunya, Departament d'Arquitectura de Computadors, Barcelona, Spain

[3]Microsoft Research Cambridge UK

{ferad.zyulkyarov, milos.milovanovic, osman.unsal.adrian.cristal, eduard.ayguade, mateo.valero}@bsc.es
{tharris}@microsoft.com

Abstract

This paper introduces an advanced hardware based approach for accelerating Software Transactional Memory (STM). The proposed solution focuses on speeding up conflict detection that grows polynomially with the number of concurrently running transactions and shared to transaction-local address resolution, which is the most frequent STM operation. This is achieved by logic split in two hardware units: Transaction Processing Core and Transactional Memory Look-Aside Buffer. The Transaction Processing Core is a separate hardware unit which does eager conflict detection and address resolution by indexing transactional objects based on their virtual addresses. The Transactional Memory Look-aside Buffer is a per-processor extension that caches the translated addresses by the Transaction Processing Core. The effect of its function is a reduced bus traffic and the time spent for communication between the CPUs and the Transaction Processing Core.

Compared with other existing solutions, our approach mainly differs in proposing an implementation that is not based on the processor cache but a separate on-chip core, uses virtual addresses, does not require application modification and is further enhanced by Transactional Memory Look-Aside Buffer. Our experiments confirm the potential of the Transaction Processing Core to dramatically speed up STM systems.

1 Introduction

The advent of shared-memory Chip Microprocessors (CMP) has created a new opening to exploit thread-level parallelism. Microprocessor manufacturers are packing more processing cores on die with each technology node. A new Moore´s law is proposed which postulates that the number of cores per

CMP will double every two years. However, programming those many-cores will be a challenge with the existing frameworks. Transactional Memory (TM) is a promising key technology for tackling this problem by abstracting some of the complexities associated with concurrent access to shared data [1]. In TM, transactions replace locking with atomic execution units, so that the programmer can focus on determining where atomicity is necessary, rather than on the mechanisms that enforce it. For example, the following code segment shows an example atomic region in a simple kernel that computes the histogram of a matrix:

```
atomic {   hist[array[i][j]]++;   }
```

With this abstraction, the programmer identifies the operations that form a critical section, while the TM implementation determines how to run that critical section in isolation from other threads.

Typical TM implementations optimistically run transactions in parallel, assuming that the transactions won't perform conflicting memory accesses, keeping tentative updated versions. When a conflict occurs, it is detected and one or more of the conflicting transactions is then aborted, undoing the tentative updates. On the other hand if the TM system determines that a transaction does not have any conflicts, the transaction can commit its tentative changes to main memory. Conflicts can be eagerly detected at each transactional read and write or they can be detected lazily at the end when a transaction wants to commit.

There are two main variants of TM, Hardware (HTM) [2][17][19][20] and Software (STM) [3][14][21]. HTM is fast but suffers from resource constraints. STM, on the other hand, is comparatively much slower but is more flexible and offers a rich expendable set of primitives. Proposals exist to deal with the HTM resource limitation problem through a Hybrid (HyTM) [4][5] approach, in which a switch can be made from HTM to STM when the hardware resources are reached, but the implementation is very complex. Most recently, there are some approaches to accelerate STM with hardware support [6][7][8] that offers the best of both worlds: an elegant, semantically rich TM that is also fast. In this paper, we also propose a hardware-accelerated STM design. However, in a significant departure from previously-proposed hardware-accelerated designs, we employ a small specialized processing core which we term the *Transactional Processing Core* (TxPC) to accelerate Software Transactional Memory.

We believe such a design approach is ideal for speeding up STM implementations. To the best of our knowledge, this is the first research that proposes a Transactional Processing Core to accelerate STM. We

do a thorough evaluation of the design space starting from the observation that the most time consuming operations in STM are the transaction verification [9][22][23], growing polynomially with increasing number of transactions in the system and the software read/writes of the tentative copies [6]. The Transactional Processing Core addresses those performance problems and accelerates STM through:

- acceleration of STM transaction validation through hardware-based eager conflict detection at object granularity;
- acceleration of STM read/writes through hardware based address resolution; in the context of transactional memory an address resolution is a translation of an object address into an address of a tentative object copy.

A practical use of the TxPC would be to solve a problem described by Shpeisman et al. [16] regarding the very attractive STM implementation that combine eager update management and lazy conflict detection [14][15]. When transactions update the objects eagerly, the subsequent reads can see the most recent value. In contrast to the lazy updates, transactions must look in a log to find the most recent update. And with the lazy conflict detection, shared objects do not keep track of their readers. But this combination has a serious problem as it allows a transaction to continue running once it is doomed to abort. Such “zombie transactions” can continue to access memory and, because they are making updates in place, they may overwrite memory locations which form parts of non-transactional data structures, or (in languags like C) make out-of-bounds array accesses, potentially leading to buffer-overflow security problems. TxPC would solve these problems by providing mechanisms to accelerate an STM using lazy update management (i.e. making updates to a transaction-private log) or eager conflict detection (i.e. detecting conflicts before they could lead to problems). These accelerations are achieved by maintaining a hardware table in TxPC with the recent mappings for heap addresses to locations in the transaction's log and offloading the conflict detection to the TxPC.

An important consideration in our baseline design of the Transactional Processing Core is to have minimal overall CMP hardware design impact, requiring no or minimal changes to the other non-specialized cores and the interconnect. Later in the paper, we develop design optimization that have more hardware design impact in return for even more acceleration. In particular, we propose Transactional Memory Translation Look-aside Buffer (TM-TLB): A TLB-like structure in every core that caches frequently used STM address translations mitigating the need to access the TxPC for those accesses.

Later, we also discuss design issues such as extending the existing core interconnect protocol versus designing dedicated hardware interconnect for the TxPC. The rest of this paper is organized as follows: In Section 2, we introduce the baseline Transactional Processing Core design and Micro Architecture. In Section 3, we explain the Architecture including ISA additions and integration with the STM library. In Section 4, we further explore the design space with and essential enhancement the Transactional Memory TLB. In Section 5, we present our results and conclude in Section 6.

2 Micro Architecture

Today, when the tendency is to host multiple cores within the same die, we propose using some of the transistors for a special purpose Transaction Processing Core (TxPC). We design TxPC as a separate hardware unit with the task to execute the slow and frequent STM operations in hardware. The STM operations that TxPC handles in hardware are eager conflict detection and address resolution. These operations are present in both types of STM systems, those that work on their own local copy of the objects and those that update objects in-place and backup the original values. The former STM systems do address resolution on each repeated read and write of an object. In the latter STM systems address resolution is not of high importance since it is done when transaction aborts and has to restore the object's buffered value. Fast eager conflict detection has the advantage of resolving conflicts at the time they happen and is crucial for both systems, especially for the second type where aborts are very expensive. To execute the aforementioned operations in hardware, TxPC indexes the transactional objects[1] in a very fast low-latency storage structure. The objects are indexed based on their virtual address and process ID which makes the TxPC to function smoothly across context switches and interrupts, issues which pose many problem for Hardware Transactional Memory systems and systems that propose mixed solutions like HyTM [4][5], VTM [10] and RTM [8]. We believe this property leads to a more implementable complexity-effective design in comparison. The transactional objects that are indexed by TxPC are explicitly provided by the STM library through a set of new ISA extensions which are described in the next section Architecture. This section focuses on the TxPC design.

To seamlessly integrate the Transaction Processing Core with the other cores on-die we add one more Transactional Memory Register (TMR) to each non-specialized core. This register will be used for storing the results that TxPC produces. So when a TxPC specific instruction is decoded by one of the non-specialized cores, it is forwarded to the TxPC. When the TxPC executes this instruction, it returns the

1 In this paper we use the term “transactional object” to describe an object that is read or written within a transaction.

result back to the associated core and this core places the result in its TMR register. The connection between the processing core and the other cores on-die can be done through the front-side bus (Figure 1 (a)) or the back-side bus (Figure 1 (b)). In our design we choose to use the front-side bus because this will require minimum architectural changes on the current processors (only changes in the bus messaging protocol), whereas a back-side bus connection will require implementing new connection ports on every core and additional interconnect. An implication of this decision is that the front-side bus could get clogged. Note that although we chose a bus-based architecture for ease of presentation, the basic TxPC micro architecture remains unchanged and could be used with other interconnect topologies such as the hypercube, 2-D or 3-D mesh or torus. Although those advanced interconnect topologies could somewhat decrease the contention due to the additional messages, the problem still remains. We tackle this problem in Section 4, and propose micro architectural enhancements that will decrease the additional strain on the core interconnect with a small additional hardware design effort.

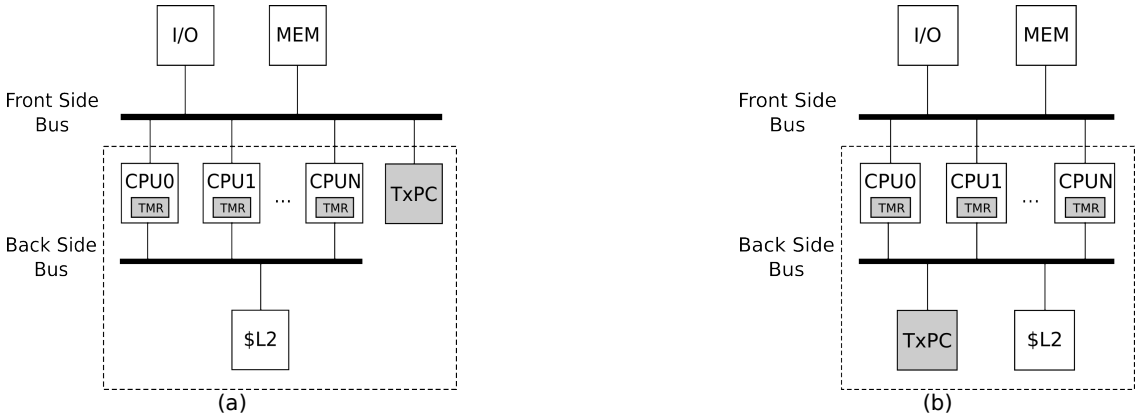

Figure 1: *Connecting the Transaction Processing Core with the other cores on-die. The gray parts represent the proposed new micro architectural extensions.*

To accelerate the most common STM operations the Transaction Processing Core indexes the objects that are accessed by the running transactions. For each shared object, TxPC maintains meta data that enables it to do conflict detection and address resolution. An abstraction of the storage structure that the TxPC uses to track a transactional object along with the meta data associated with each object is shown in Figure 2.

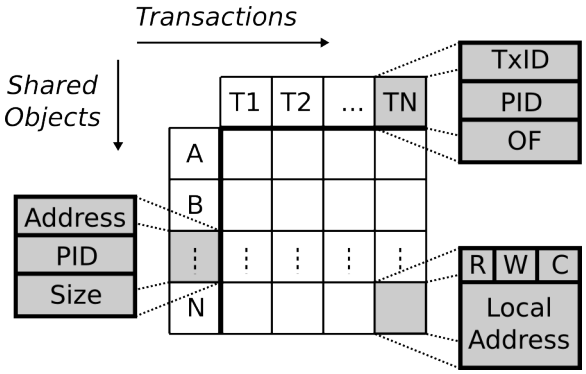

Figure 2: *The Transaction Processing Core uses a matrix like data structure to index the transactional objects. The big grayed boxes show the details about the associated group of cells. Each row contains information about a transactional objects. Transactional objects are distinguished by their virtual address and process ID. Each column contains information about the objects that a given transaction has read or written. Different transactions are distinguished by the transaction ID (TxID) and the process ID (PID). Every transaction is associated with an overflow (OF) bit that tells whether the read and write set of the transaction has overflowed. Each cell in the matrix tells whether a transaction has read (R) the object, written (W) the object, the object creates a conflict (C) within the transaction and the transaction-local address where the copy of the object is stored.*

This structure is implemented in the TxPC through a doubly indexed CAM structure, using transaction ID (TxID) and object address as the indexes. However, in Figure 2, we use a matrix like storage representation to facilitate the discussion. Each row of the matrix represents an object that has been accessed by any transaction regardless of the transaction is tracked by the TxPC or by the STM library. To make the management of the transactional objects as simple and efficient as possible, we base our design on the very crucial property which states that objects tracked in the hardware are not tracked in software and vice versa (the two set of objects are disjoint). Because the processing core can be used by multiple programs, different objects are distinguished by both their virtual address and the process ID. To make the common case efficient, the processing core tracks the read and write set of a limited number of transactions. Each column in the matrix in Figure 2 represents the read and write set of a transaction. Transactions that are tracked by TxPC are identified by the process ID (PID) and the transaction ID (TxID). The need for associating each transaction with the process ID, comes out of the fact that multiple programs can run transactions at the same time and if they have the same TxID, then they are distinguished by the PID. Last but not least, each transaction is associated with an overflow bit (OF). The overflow bit is set to 1 if the read or write set of the transaction that is tracked by TxPC has overflowed, meaning that part of the read or write set of the transaction resides in hardware and part in software. When the read or write set of a transaction overflows, then it is split between the software and hardware in a way that none of these two sub-parts are overlapping meaning that no object is both in software and hardware. Each cell in the matrix contains information that tells whether the object is in transaction's read set (R), whether the object is in transaction's write set (W), the object is a source of conflict (C), and

the address (Local Address) where the transaction has stored a private copy of the object for its own use[2]. Here the C bit requires a little bit more explanation as it is used to facilitate fast (few cycles) validation. When TxPC detects that reading or writing a particular object creates a conflict, the C bit of the cell is set to 1. Later on, when TxPC does validation for an accelerated transaction, it applies an “or” operation on all C bits in the transaction's hardware read-write set and if the result is 0 then there is no conflict, if the result is 1 there is a conflict.

When the processing core indexes the shared objects and the transactions in the system, it always tries to fill any empty space in the table from Figure 2. For example, when an object is accessed for the first time in a transaction, TxPC tries to index it in one of the rows in the table and when a new transaction starts, TxPC tries to index it into one of the empty columns in the table from Figure 2. When a new object is requested for opening by the STM library, it is indexed into one of the empty rows in the mapping table. Because of the restricted storage space, the processing core may exhibit two types of overflow: object overflow and transaction overflow. Object overflow occurs when there is no available row to track new objects. Figure 3 visualizes an object overflow, when transaction T1 reads object B and there is not any empty row in the table. In this case the overflow bit of the transaction (OF) is set to 1 and the object is indexed in the software by the STM library. After this moment, transaction's read-write set is split across the hardware and software and TxPC can serve only the instructions that are related with the read-write set located in the hardware (i.e. to validate only the hardware part which is the object A) and the operations related to the objects in the software are handled by the STM as it was done before. Transaction overflow happens when there is no more columns in the table from Figure 2 to track the reader and writer transactions of the object. In this case, when the STM library wants to start a new transaction and the table is full, the TxPC sets the TMR register status bit to FAIL instead of SUCCESS. Besides trying to start a new transaction when the Transaction Processing Core's table is full, an transaction overflow may also occur when mixing accelerated[3] and non-accelerated transactions in a program. An example is shown in Figure 4.

2 For the STM libraries that do in-place update, the Local Address is the place where the transaction backed-up the original value of the object before updating it.

3 We use the term accelerated transaction to distinguish the transactions that are tracked by the processing core from those that are not tracked.

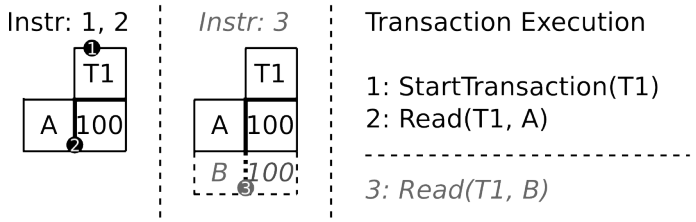

Figure 3: *Object overflow. The figure visualizes an indexing table with dimensions 1x1. Object overflow happens when a transaction wants to access an object when there is no empty row to track it.*

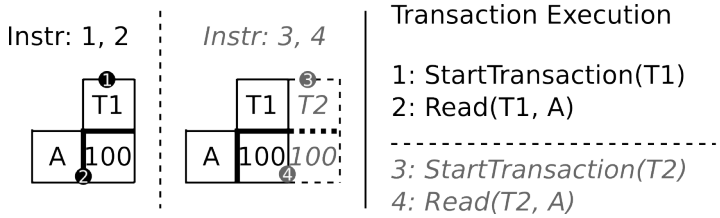

Figure 4: *Transaction overflow. The figure visualizes an indexing table with dimensions 1x1. Transaction overflow happens when there is no empty columns on the table to track the readers and writers of an object.*

In Figure 4, let´s assume that the Transaction Processing Core can track only one transaction. The first instruction starts a new transaction. After a transaction T1 is started, it is registered into the table. Then T1 reads object A that is also registered into the table. Later transaction T2 starts. The processing can not index T2 and it is indexed by the STM. Next follows the problematic operation when T2 wants to read object A. The indexing table in the processing core is already full and the processing core cannot later mark T2 as a reader of object A. To obey the rule for managing the objects across hardware and software and preserve the set of objects tracked by STM library and TxPC disjoint, is necessary either to evict object A to the software or somehow to mark that T2 is a reader of A in hardware. The option to evict A into software is not desirable because it will cause all its readers and writers to split their read-write set across the hardware and software even if there are empty rows in the table for tracking new objects. In this case we decide to append a pointer field to the CAM structure from Figure 2. The pointer points to a reserved place in main memory used to track the reader and writer transactions like T2 into that extended place. Figure 5 shows the extended version of the indexing table from Figure 2.

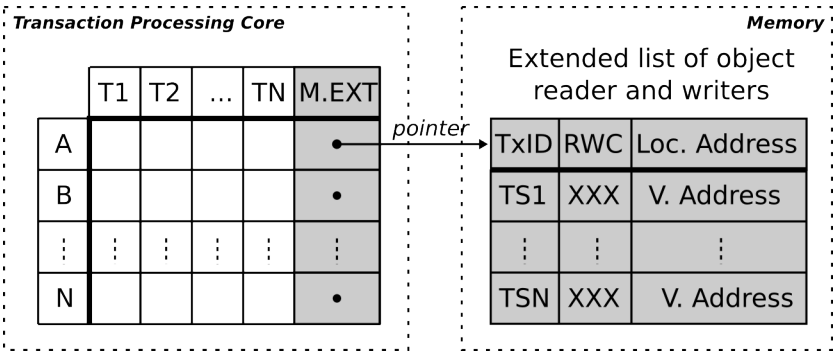

Figure 5: *Extended indexing table. The gray colored parts in the figure represents the extensions. Evey object is extended with a pointer to a reserved place in memory, where its non-accelerated reader and writer transactions are listed. In the figure object A has reader and writer transactions that are not accelerated by the processing core and are indexed in the memory.*

The gray colored parts in Figure 5 represents the introduced extensions. Every object is extended with a virtual address pointer to a reserved place in the main memory where its non-accelerated reader/writer transactions are indexed. With this structure the processing core still makes the common case when there is no overflow fast and the general case with an overflow slow but yet efficient. As shown in Figure 5, object A has non-accelerated reader/writer transaction and its extension pointer points to a reserved place in memory where these transactions are indexed. The meta data stored on the table extension is the transaction ID that accessed the object, the access bit mask, and the transaction local address of the object. The extension pointer of the objects that don't have non-accelerated readers or writers is set to NULL.

When a transaction commits, aborts or retries, all the object entries associated with that transaction have to be updated and those objects that do not have any readers or writers should be unregistered. The cleaning process for an accelerated transaction is trivial and immediate as all the cells in its column (see Figure 2) are flash cleared in a single cycle. The cleaning process for non-accelerated transactions is a little bit complicated because traversing the memory would impact on the performance of the TxPC. To avoid this slow-down we propose a delayed cleanup by a special low priority thread that runs in TxPC and does garbage collection. In this way, when a non-accelerating transaction terminates, it will be marked in the TxPC and the garbage collector thread will deal with the memory deallocation. After the cleaning process, if there are deleted objects from the table, the STM is promoted to put in, objects that are tracked in software. In this way the processing core is utilized most efficiently.

3 Architecture

To use the Transaction Processing Core for accelerating STM library we extend the current processors'

instruction set with 9 new instructions and 1 Transactional Memory Register (TMR). The STM library should use these new instructions to explicitly ask TxPC do address resolution and validation and provide TxPC with information about the transactional objects it accesses. TMR register is the place where the program looks for the results that TxPC returns.

The integration of TxPC with the STM library is trivial as it requires to change the implementation of few library's interface functions. The existence of the TxPC leaves invisible to the programmer and it does not require changes in the TM applications.

3.1 ISA Extension

To benefit from the functionality of the Transaction Processing Core, we extend the current processors' ISA with 9 new instructions and one Transactional Memory Register (TMR). The purpose of the new ISA extension is to provide the STM library a mechanism to do conflict detection, validation and address resolution in hardware and to tell the Transaction Processing Core about its intention to open objects for reading and writing. The new instructions are summarized on the table below.

Instruction	Description
OnRead	Resolves the shared object's address into transaction local address from the transaction's read set.
OnWrite	Resolves the shared object's address into transaction local address from the transaction's write set.
OnValidate	Validates the transaction.
OnStart	Creates and initializes a transaction entry.
OnCommit	When transaction commits, unregisters a transaction if it is tracked by the Processing Core.
OnAbort	When transaction aborts, unregisters a transaction if it is tracked by the Processing Core.
OnRetry	Cleans the transaction's read and write set.
OpenObjectForRead	Registers an object into the transaction's read set.
OpenObjectForWrite	Registers an object into the transaction's write set.

Table 1: *Transaction Processing Core ISA extension.*

`OnRead TxID sharedObjectAddress`

`OnRead` instruction resolves the shared object's address into transaction local address from the transaction's read set. If the the address can be resolved it is placed in TMR register; if the address cannot be resolved the contents of TMR is set to FAIL.

`OnWrite TxID sharedObjectAddress`

OnWrite instruction resolves the shared object's address into transaction local address from the transaction's write set. If the the address can be resolved it is put in TMR register; if the address cannot be resolved the contents of TMR is set to FAIL.

OnValidate TxID

OnValidate instruction validates the transaction. The result of validation is put into TMR register and it could be:

- SUCCESS if the transaction is valid and there is no overflow;
- SUCCESS_OVERFLOW if transaction's read/write set tracked by the TxPC is valid but it overflows to the software and the STM has to validate the overflowed part explicitly;
- FAIL if the transaction is not tracked by the TxPC; in this case STM has to do the validation.

OnStar

OnStart instruction creates and initializes a transaction entry. The result of this instruction is put into TMR register and it could be:

- SUCCESS if the transaction is registered successfully;
- FAIL the Processing Core has overflowed and can not track this transaction; this can be used as a hint for scheduling transactions in STM.

OnCommit TxID

This instruction unregisters a tracked instruction by the Transaction Processing Core when it commits.

OnAbort[4] TxID

This instruction unregisters a tracked instruction by the Transaction Processing Core when it aborts.

OnRetry TxID

To be called before STM retry. If the transaction is already tracked by the Transaction Processing Core this instruction cleans its read/write set, if the transaction is not tracked and it can be tracked it is registered (has the effect of OnAbort and OnStart called subsequently). The result of validation is put into TMR register and it could be:

- SUCCESS if the transaction is registered successfully;
- FAIL the Processing Core has overflowed and can not track this transaction; this can be used as a hint for scheduling transactions in STM.

OpenObjectForRead TxID sharedObjectAddress localAddress objectSize

This instruction registers an object into the transaction's read set. It must be invoked every time when an object is being read for the first time since the transaction execution. The execution of this instruction

4 OnCommit and OnAbort do the same thing but for not they are distinguished because of consistency reasons and for the future because of extension reasons if the Processing Core is added a functionality to abort and commit a transaction.

is asynchronous and the default value is optimistically assumed to be SUCCESS (no conflict). In case its execution result to a conflict, the STM library is acknowledged about that with an interrupt raised by the TxPC. The interrupt contains the transaction ID and the object that is the source of the conflict. The asynchronous nature of this instruction has the advantage of executing transactions parallel with the conflict detection.

OpenObjectForWrite TxID sharedObjectAddress localAddress objectSize

This instruction has the same semantics os OpenObjectForRead but it registers an object into the transaction's write set.

3.2 Integration with STM

We use the Transaction Processing Core to accelerate the Nebelung STM library [12][14]. The Nebelung STM library implements lazy update management and lazy conflict detection. The external interface of the library is given in Figure 6.

```
Transaction*   createtx();
void           destroytx(Transaction *t);
void           starttx(Transaction *t);
status         committx(Transaction *t);
void           aborttx(Transaction *t);
void           retrytx(Transaction *t);
void*          readtx(Transaction *t, void *addr, int blockSize);
void*          writetx(Transaction *t, void *addr, void *obj, int blockSize);
void*          invalidateAddr(Transaction* t, void* adr)
status         validatetx(Transaction *t);
status         resolveConflicttx(Transaction *t);
void           _mCommit(Transaction *t);
```

Figure 6: *Nebelung STM library interface functions. Functions `createtx`, `destroytx`, `starttx`, `committx`, `aborttx` and `retrytx` are self explanatory. These functions create, destroy, start, commit and abort the transactions respectively. Functions `readtx` and `writetx` are called whenever transaction accesses some memory location. Function `invalidateAddr` is used to remove function local variables from STM system when the function is finished. Functions `validatetx`, `resolveConflicttx` and `_mCommit` are identified by the decomposition of the `committx` function and they can be parallelized and accelerated is software.*

Functions `createtx`, `destroytx`, `starttx`, `committx` and `aborttx`, `retrytx` respectively create, destroy, start, commit, abort and re-execute a transaction. Functions `readtx` and `writetx` are called whenever transaction accesses some memory location. Function `invalidateAddr` is used to remove transaction local variables from STM system when the function is finished. Functions `validatetx`, `resolveConflicttx` and `_mCommit` are identified by the decomposition of the `committx` function.

Making the STM library work with the Transaction Processing Core is trivial. The user applications are not affected and the TxPC integration requires minor changes within the implementation of the library interface functions. Unlike most other hardware-accelerated STMs which require user source code

modifications, a big advantage of the TxPC architecture is that it requires only the STM library to be recompiled. The functions that should be changed and the changes that should be done in order to accelerate transactions are shown in the table below. The changes are inlined assembler code that calls the extended ISA instructions. The STM first tries to execute the operation in the TxPC, if TxPC fails then it executes in the software as it was doing originally. Table 2 shows the required changes that should be done in STM's interface implementation.

```
void starttx (Transaction *t)
{
  int TxPC_res;

   asm("OnStart %1; movl %%TMR, %0"
        : "=r" (TxPC_res) : "r" (t->id):");

  if (TxPC_res == FAIL) schedule_transaction(t);

   /* Continue the original core */
}
```

```
status committx (Transaction *t)
{
   /* ... The original code until here */

   if (status == SUCCESS)
      asm("OnCommit %0": : "r" (t->id):");

   return status;
}
```

```
void retry (Transaction *t)
{
   asm("OnRetry %0": : "r" (t->id):");

   /* Continue the original core */
}
```

```
void aborttx (Transaction *t)
{
   asm("OnAbort %0": : "r" (t->id):");

   /* Continue the original core */
}
```

```
void*  writetx (Transaction *t, void *addr, void* obj, int
blockSize)
{
  void* local_addr = NULL;

  asm("OnWrite %1, %2; movl %%TMR, %0";
       :"=r" (local_addr) , "r" (t->id) , "r" (addr):);

  if (local_addr == FAIL)
  {
     local_addr = resolve(addr);
     if (local_addr == NULL)
     {
         local_addr = create_local_copy(addr);

         asm("OpenObjectForWrite %0, %1, %2, %3"
                : : "r" (t->id),
                    "r" (addr),
                    "r" (local_addr),
                    "r"(size) : );
     }
  }

  /* Continue the original core */
}
```

```
void*  readtx (Transaction *t, void *addr, int blockSize)
{
  void* local_addr = NULL;

  asm("OnRead %1, %2; movl %%TMR, %0";
       :"=r" (local_addr) , "r" (t->id) , "r" (addr):);

  if (local_addr == FAIL)
  {
     local_addr = resolve(addr);
     if (local_addr == NULL)
     {
         local_addr = create_local_copy(addr);

         asm("OpenObjectForRead %0, %1, %2, %3"
                : : "r" (t->id),
                    "r" (addr),
                    "r" (local_addr),
                    "r" (size) : );
     }
  }

   /* Continue the original core */
}
```

```
status validatetx (Transaction *t)
{
   status res;

   asm("OnValidate %0": "=r" (res): "r" (t->id) : );

   if (res == SUCCESS) return SUCCESS;
   else if (res == SUCCESS_OVERFLOW)
   {
       status stm_res = stm_validate_overflowed_part(t);
       if (stm_res == SUCCESS) return SUCCESS;
   }
   else if (res == FAIL)
      return stm_validate_all_read_write_set(t);
}
```

Table 2: *The required source code changes in the STM library implementation in order to integrate the Transaction Processing*

Core with the STM library.

4 Transactional Memory TLB

In Software Transactional Memory systems that work on the private copies of the shared objects, translation from shared object address to transaction-local address is a very frequent operation that is done in every read and write STM operation. Our experiments that follow in the next section also confirm that address resolution is an intensively repeating operation (Table 6 and Table 7). We foresee that many address resolution operations forwarded from different CPUs to the Transaction Processing Core will create excess bus traffic and flood the TxPC. To deal with this issue, we propose to add a small fully-associative cache in every core that will store the already resolved addresses by the Transaction Processing Core. We name this cache as Transactional Memory Look-aside Buffer (TM-TLB) because its purpose is quite similar to the Translation Look-aside Buffer (TLB) to cache address mappings. The TM-TLB cannot completely sustain context switches or interrupts, so it may have to be flushed. When a context switch or interrupt happens, it is not guarantied that when a process is resumed it will be assigned to the same CPU it was executing on before. But if the OS assigns the process to the same CPU then transactions can reuse the TM-TLB entries if they have not been deleted.

The way how TM-TLB is involved in address translation is shown in Figure 7. When CPU decodes `OnRead` or `OnWrite` instruction it first checks the TM-TLB for a cached address. If there is a hit, the CPU sets the value of the TMR register with the transaction-local address without forwarding the operation to the TxPC. In this case we save a time for communication between the CPU and the TxPC and do not create bus traffic. But if there is a miss, then the CPU forwards the instruction to TxPC, TxPC computes the transaction-local address, returns it to the CPU, the CPU caches the translation into the TM-TLB and sets the TMR register with the resolved address. In case that TxPC cannot resolve the address because of the object is not tracked in hardware, TM-TLB cache it as FAIL and returns FAIL to the subsequent attempts to resolve the same address. Caching of the object addresses can be done not only through the `OnRead` or `OnWrite` instructions but at the time when an object is being opened with one of the the `OpenObjectForRead` or `OpenObjecForWrite` instructions. In this case, even if TxPC cannot index the new object, TM-TLB can cache it if there are empty cache slots.

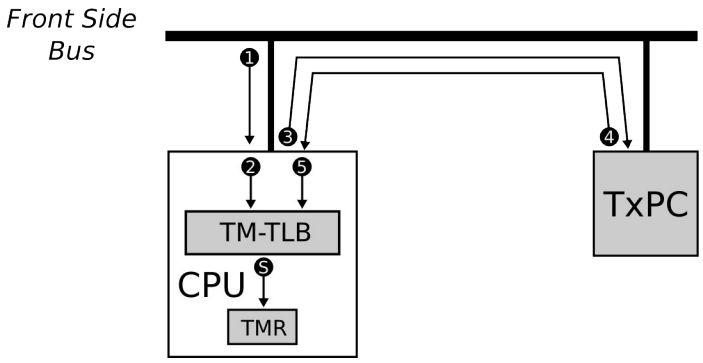

Figure 7: *This figure shows how address resolution is done with the TM-TLB cache. When a resolve (`OnRead`/`OnWrite`) instruction decoded (1), the CPU checks the TM-TLB cache (2). If there is a hit the result is stored into TMR register (S). If there is a miss, the resolution instruction is forwarded to the TxPC (3), the result that the TxPC generates is returned back to the CPU (4). The CPU caches the result into the TM-TLB (5) and puts the result into the TMR register (6).*

When a transaction terminates (i.e. `OnAbort`, `OnCommit` or `OnRetry`), the cleaning of the TM-TLB can be ignored or the CPU can clean all the entries associated with the particular transaction. In the first case, when another transaction is started the replacement algorithm will clean the dirtiness in a natural way, but this may result in evicting an entry that belongs to a running transaction instead of an already terminated transaction. Therefore in our design we prefer to clean TM-TLB at the time when transaction terminates.

An interesting issue that deserves attention is the case when both TM-TLB and TxPC cannot resolve an address and the address resolution must be done in software. This situation may occur after a context switch or cache eviction. In this case all the address resolutions for that particular object must be done in software. Reading or updating this object in a loop may have a significant impact on the performance. This penalty can be compensated for the cost of breaking the encapsulation in the design and making the STM library aware of TM-TLB. One solution for this problem is adding a new instruction that STM library may use to force TM-TLB to cache a resolved address in software (not the TxPC).

5 Experiments

To estimate the potential performance gains that can be achieved by using the Transaction Processing Core, we use a simplified software model of the processing core. The simplified model implements and simulates the data structure shown in Figure 2 with a preset values for the number of objects and transactions that could be tracked in hardware. We also assign a cost for executing each of the instructions in Table 1. The experiments are performed by first running the transactional workloads without the support of TxPC and count how many CPU cycles is spent for the execution of the STM

operations that TxPC can accelerate. Then we run the same workloads on the STM library accelerated by the TxPC software model and count how many of the STM operations can be handled in by the evaluation model. Every STM operation that TxPC can handle we consider as a speedup and then calculate the total speedup over the complete execution of the workloads. We compute the total speedup as a sum of all per-instruction speedups. Next in this section we describe in more details about workload applications we use, the setup for the experiments and discuss the obtained results.

5.1 Transactional Memory Workloads

To evaluate the performance of the Transaction Processing Core we use two small workload applications: Bank Account Workload and BPlusTree. The Bank Account workload is a very simple application that atomically debits money from one account and deposits money to another account. This application represents a scenario where few shared objects are a source of a high contention.

B+ tree application simulates transaction execution on the complex and very large data structures and tests the composability of the transactions. We implemented B+ tree structure and the *get*, *put* and *lookup* operation using transactions. Then we created function *move* which moves the value from one B+ tree to the other atomically using get, put and lookup function and nesting of transaction. We filled initially the B+ trees with large amount of data (1M-100M of values) and then performed the atomic moves concurrently with lookups. Additional parameter we tested was the calculation time in the transaction. We realized that this is very important parameter, because the calculation is the part which can be parallelized. So we did the tests in the following way: function calculate takes the data from one B+ tree then sleep for 50μs (this time simulates the calculation) and then puts the data in the other B+ tree. Data structures are large, read and write sets are large and therefore the probability of a conflict is low.

5.2 Evaluation Model

To evaluate the Transaction Processing Core we use a simplified software model that approximates the design defined in Section 2 and functionality defined by the ISA extension in Section 3.1. The experimental model uses the identical data structure shown in Figure 2 to index the transactional objects. The dimensions of the whole data structure and memory required to store every meta data recorded in the index table is shown in Figure 8 and summarized in Table 2.

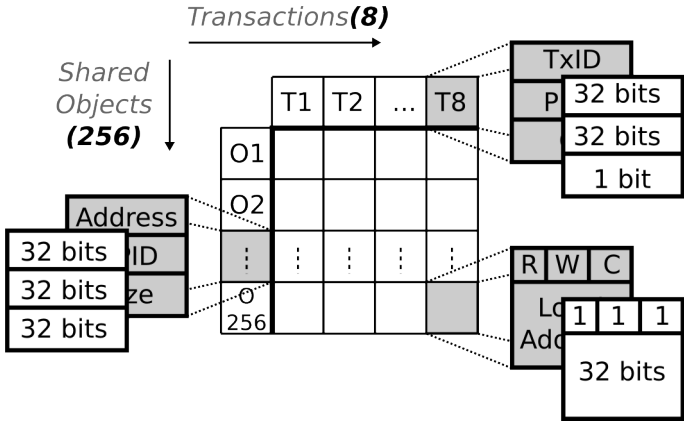

Figure 8: *Instantiation of the table from Figure 2. The experimental model of the Processing Core handles up to 8 transactions in and 256 objects. The big gray boxes show the details about the cell in the table and the white big box next to each gray big box show the size of each tag in bits.*

The evaluation model of the TxPC abstracts can host up to 256 different objects and 8 transactions. Being able to track 8 transactions we model an 8 core CPU. Each object header uses 32 bits for storing the object's address, 32 bits for storing process ID 8 bytes per object header. A transaction header uses 32 bits for transaction ID, 31 bits for process ID and 1 bit for OF tag, which sums to 5 bytes per transaction. Each cell uses 1 bit per R, W and C tags and 32 bits to store the object's transaction-local address which sums to 5 bytes[5]. The total memory demand of the indexing data structure is 12 328 bytes (12.04KB). The memory use of the indexing table in the TxPC model is summarized in Table 2.

Structure	Memory	Num. of Structures	Total Memory
Object header	6 bytes	256	2048 bytes
Transaction header	5 bytes	8	40 bytes
Cell	5 bytes	2048	10 240 bytes
Total			***12 328 bytes = 12.04KB***

Table 3: *Summary of the memory demand by the TxPC evaluation model.*

To make the evaluation model of TxPC complete, we assign every TxPC instruction from Table 1 cost in terms of CPU cycles. We consider that assigned CPU cycle values are meaningful in the context of current micro architecture trends. The cost of each instruction is given in Table 4.

5 We round the it to form a whole byte.

Instruction	Cost
One way inter-processor communication	5 cycles
OnRead	2 cycles
OnWrite	2 cycles
OnValidate	2 cycles
OnStart	2 cycles
OnCommit	2 cycles
OnAbort	2 cycles
OnRetry	2 cycles
OpenObjectForRead	2 cycles
OpenObjectForWrite	2 cycles

Table 4: *The cost of for executing TxPC instructions*

We assume that it takes 10 cycles for inter-core communication (forwarding an instruction and receiving the result) when there is no bus contention and every instruction takes 2 cycles to execute in TxPC when there is no object overflow. In case of object overflow instructions `OnRead`, `OnWrite` and `OnValidate` will execute longer because TxPC will have to visit the main memory. Based on the evaluation TxPC model here, the next subsection 5.3 examines the impact of the TxPC over 2 TM applications described in the previous subsection 5.1.

5.3 Experiments and Results

We run our experiments in a real SMP machine that has 4 dual core CPUs each 3.2 GHz and total of 16GB RAM. The experiment consists of two parts: first running the workloads without hardware acceleration and second with hardware acceleration. The purpose of the first part of the experiment is to capture the runtime characteristics of the STM operations, such as how many CPU cycles it takes to execute any of the STM operations in the experimental computer system. The purpose of the second part of the experiment is to determine how many STM operations can be handled by the Transaction Processing Core and consequently what would be the potential speedup of the TxPC. In both parts of the experiments we execute the two workload applications 4 times with 2, 4 and 8 threads. Then we take the average of the all results.

Tables 5 and 6 summarize the average cost for each STM operation in CPU cycles and Tables 7 and 8 summarize how many times each STM operation is invoked in the Bank Account and BPlusTree applications respectively.

Operation	2 Thrds	4 Thrds	8 Thrds
read	4058	3853	3625
write	1746	1738	1707
validate	3290293643	3916723364	617739258

Table 5: *The average cost of STM operations for the Bank Account application in CPU cycles.*

Operation	2 Thrds	4 Thrds	8 Thrds
read	1887	2079	2499
write	2995	3529	4563
validate	94232	905835	1746197

Table 6: *The average cost of STM operations for the BPlusTree application in CPU cycles.*

Operation	2 Thrds	4 Thrds	8 Thrds
read	154	182	238
write	22	26	34
validate	11	13	17

Table 7: *The total number of executions of the STM operations in Bank Account application.*

Operation	2 Thrds	4 Thrds	8 Thrds
read	287064	577239	1170723
write	23138	46231	93228
validate	400	800	1600

Table 8: *The total number of executions of the STM operations in BPlusTree application.*

From tables 5, 6, 7 and 8 we conclude that the cost for reads and writes is approximately constant and does not depend neither of the application characteristics nor the number of transactions. When the number of the reads increase the initialization overheads are amortized and the average computation time is reduced. But this is not true for the validation. The time spent in validation depends on both the application characteristics and the number of threads that execute transactions. With the Bank Account application, although the very few reads and writes the validation time is too high compared to the BPlusTree because of the high contention. Also, the total number of reads and writes depends on the workload application if the transaction blocks are big or small.

Tables 9 and 10 show the number of the STM operations executed by the TxPC and tables 11 and 12 show maximum achievable per STM operation speedup when workload applications are accelerated by the TxPC. The speedup is calculated over the sum of execution time for each type of operation as the cost for every TxPC instruction is taken from Table 4.

Operation	2 Thrds	4 Thrds	8 Thrds
read	154	182	238
write	22	26	34
validate	11	13	17

Table 9: *The number of STM operations that are executed by the TxPC for the Bank Account application.*

Operation	2 Thrds	4 Thrds	8 Thrds
read	287064	575737	1169437
write	23138	46231	93123
validate	400	791	1578

Table 10: *The number of STM operations that are executed by the TxPC for the BPlusTree application.*

Operation	2 Thrds	4 Thrds	8 Thrds
read	338.17	321.08	302.08
write	145.5	144.83	142.25
validate	274191137	326393614	51478271

Table 11: *The per-operation speedup for the Bank Account application.*

Operation	2 Thrds	4 Thrds	8 Thrds
read	157.25	119.63	169.63
write	249.58	294.08	266.44
validate	7852.67	88.79	59.24

Table 12: *The per-operation speedup for the BPlusTree application.*

The results for the Bank Account application show that TxPC can execute all the STM operations regardless the number of the threads. Therefore the expensive validate operation is executed immediately. The results for the BPlusTree application show that when executing 2 threads all the operations are handled by the TxPC and when the number of threads increase the number of the operations that TxPC can handle decreases. The cause for this to happen is that the different threads open too many objects that cannot fit in the hardware. But although the decrease in the number of objects, TxPC can handle at worst case up to 99% of the address resolutions, %98 of validations. The speedup of the validation in BPlusTree also decreases when the number of the threads increase. The rate of decrease is quite big because our simplified evaluation model of TxPC does not validate the transactions which read-write set overflows. But the complete TxPC which design is described in Section 2 is capable to validate the part of the transaction's read-write that is in hardware and then STM validates the part that is in software. The speedup of the address resolution, done in every read and write operation is constant for the application.

As a last word on the preliminary experimental result, we conclude that TxPC would be highly efficient for detecting conflicts in TM applications that are written with coarse grain atomic blocks and doing address resolution in applications that repeatedly operate on the same set of data as the scientific applications do.

6 Scaling with Multiple TxPC Cores

In the proposed architecture so far, utilizing only a single TxPC core would be a source of bottleneck when the number of regular cores on-die becomes considerably big (i.e. 16 or 32 cores). Determining the exact number of regular cores that would saturate the system is currently out of the scope of this research work. To solve the encountered scalability problem we propose using multiple TxPC cores on-die when a single TxPC does not suffice. Because TxPC is a self contained on-die unit, integrating many TxPCs would not require any architectural re-design of our system, but only connecting them to the

interconnect (in our case the bus). What is necessary, is to define only the way how the transactional load will be distributed among the multiple cores. We propose two different policies for distributing the load among the cores:

- based on address clustering; and
- based on thread clustering.

In the first policy, the address space is divided by the number of the TxPC cores and each TxPC is assigned a range of addresses that it is responsible to track. In the second policy, each TxPC core is assigned threads that it should serve.

When using address clustering, all the TxPC cores examine the address of the transactional instruction placed on the bus and the TxPC which assigned address range has the address executes the instruction and the other TxPCs ignore it. When using thread clustering, all the TxPC cores examine the thread ID of the transactional instruction and the TxPC that is assigned the corresponding thread, executes the instruction. In the both policies, to detect a conflict, all the cores are involved regardless the range of addresses or the thread ID. To check for conflict, every TxPC examines the read/write set of the transaction and in case of conflict respectively rises an exception if the instruction is OpenObjectForRead/OpenObjectForWrite or sets the RTM register of the relevant core to FAIL (instead of SUCCESS or SUCCESS_OVERFLOW) if the instruction is OnValidate.

We leave the performance evaluation of multiple TxPC cores as another research work, as thecurrent paper focuses mainly on accelerating transactional applications with a single TxPC.

7 Conclusion and Future Work

Transactional Memory is a concurrency control mechanism that promises to increase utilization of Chip-Multiprocessors by exploiting Thread Level Parallelism. There exists two distinct implementation for Transactional Memory: Software Transactional Memory and Hardware Transactional Memory. STM is flexible, not limited in memory space and time, but is slow. HTM is fast, but limited in memory space and time.

In this paper we proposed, Transaction Processing Core and Transactional Memory Look-aside buffer as an integral approach for accelerating the slow and frequent STM operations in hardware. TxPC is designed as a separate core on-die that does eager conflict detection and address translation from shared global address to transaction local address. TM-TLB is a per-processor buffer that caches the

already resolved addresses by TxPC and this way saves inter-core communication and bus traffic. The integration of TxPC into the existing STM libraries is trivial as it requires to do minor changes in the library's interface implementation and does not require any change in the TM applications. TM-TLB is invisible to the STM library and TxPC.

The experiments that we did with the TxPC's software simulation model, confirmed our expectations that the common case in executing TM applications can be handled by TxPC resulting to a tremendous runtime speedup compared to the STM. The TM applications that have coarse grain atomic blocks would benefit from eager conflict detection and TM applications that do repeated operations on a small number of objects like the scientific applications, would benefit from fast address resolution which is involved in every object reading or writing.

During our experiments with the STM library and TxPC, we noticed that the validation process is rather parallel and can be done in parallel for different transactions. As a future work we will investigate the possible solutions that involve multiple TxPC cores on-die. Using multiple Transaction Processing Cores seems to have a lot of benefits but on the other side involves serious problems that should be resolved such as synchronization.

References

[1] J. Larus and R. Rajwar, “Transactional Memory”, Morgan Claypool, 2006.

[2] M. Herlihy and J. E. B. Moss, “Transactional Memory: Architectural Support for Lock-Free Data Structures”, 20th International Symposium on Computer Architecture, May 1993.

[3] N. Shavit and D. Touitou, "Software Transactional Memory", 14th Annual ACM Symposium on Principles of Distributed Computing, August 1995.

[4] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir and D. Nussbaum, "Hybrid Transactional Memory", 12th International Conference ovn Architectural Support for Programming Languages and Operating Systems, October 2006.

[5] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, A. Nguyen, “Hybrid Transactional Memory”, ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming , March 2006.

[6] B. Saha, A.-R. Adl-Tabatabai, Q. Jacobson, “Architectural Support for Software Transactional Memory”, In 39th International Symposium in Microarchitecture, December 2006.

[7] A. Shriraman, M. F. Spear, H. Hossain, V. Marathe, S. Dwarkadas, and M. L. Scott, “An Integrated Hardware-Software Approach to Flexible Transactional Memory”, 34th International Symposium on Computer Architecture (ISCA), June 2007.

[8] A. Shriraman, V. J. Marathe, S. Dwarkadas, M. L. Scott, D. Eisenstat, C. Heriot, W. N. Scherrer, M. F. Spear “Hardware Acceleration of Software Transactional Memory”, 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT), June 2006

[9] C. Perfumo, N. Sonmez, O.S. Unsal, A. Cristal, M. Valero and T. Harris, “Dissecting Transactional Executions in Haskell”, 2nd ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT), August 2007.

[10] R. Rajwar, M. Herlihy, K. Lai, “Virtualizing Transactional Memory”, 32nd Annual International Symposium on Computer Architecture (ISCA), June 2005

[11] W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M. Van Biesbrouck, G. Pokam, B. Calder, O. Colavin, “Unbounded page-based transactional memory”, Proceedings of the 2006 ASPLOS Conference, October 2006

[12] M. Milovanović, O. S. Unsal, A. Cristal, S. Stipić, F. Zyulkyarov and M. Valero, “Compile time support for using Transactional Memory in C/C++ applications”, 11th Annual Workshop on the Interaction between Compilers and Computer Architecture INTERACT-11, Phoenix, Arizona, February 2007.

[13] M. Milovanović, R. Ferrer, O. S. Unsal, A. Cristal, X. Martorell, E. Ayquadé, J. Labarta and M. Valero, ¨Transactional Memory and OpenMP¨, IWOMP 2007, Beijing, China, Jun 2007.

[14] B. Saha, Adl-Tabatabai , R. L. Hudson , Chi Cao Minh , Benjamin Hertzberg, “McRT-STM: a high performance software transactional memory system for a multi-core runtime”, Proceedings of the eleventh ACM SIGPLAN (PPoPP), March 2006

[15] T. Harris, M. Plesko, A. Shinnar, D. Tarditi, “Optimizing Memory Transactions”, Proceedings of the 2006 PLDI Conference (2006), June 2006

[16] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai1, S. Balensiefer, D. Grossman, R. L. Hudson, K. Moore, and B. Saha. Enforcing Isolation and Ordering in STM. In the Proceedings of the 2007 Conference on Programming Language Design and Implementation (PLDI), March 2007.

[17] M. P. Herlihy and J. E. B. Moss, “Transactional Support for Lock-Free Data Structures”, Technical Report 92/07, Digital Cambridge Research Lab, December 1992

[18] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, “Unbounded Transactiona Memory”, (HPCA), Feb. 2005

[19] K.E. Moore, J. Bobba, M.J. Moravan, M.D. Hill, and D.A. Wood. “LogTM: Log-based transactional memory” In Proc. 12th Annual International Symposium on High Performance Computer Architecture (HPCA), 2006

[20] L. Hammond, V. Wong , M. Chen , B. D. Carlstrom , J. D. Davis , B. Hertzberg , M. K. Prabhu , H. Wijaya , C. Kozyrakis , K. Olukotun, “Transactional Memory Coherence and Consistency”, Proceedings of the 31st annual international symposium on Computer architecture, p.102, June 2004

[21] Adl-Tabatabai, A. Lewis, B.T. Menon, V.S. Murphy, R.B. Saha and Spheisman T., “Compiler and Runtime Optimizations for Efficient Software Transactional Memory”, PLDI 2006

[22] J Babba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, D. A. Wood, “Performance Pathologies in Hardware Transactional Memory”, Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA), June 2007

[23] J. Chung, H. Chafi, C. Minh, A. Mc Donald, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun 12th International Symposium on High Performance Computer Architecture (HPCA), Austin, Texas, USA, 11-15 February 2006.