Hardware Transactional Memory with Software-Defined Conflicts

Rubén Titos-Gil[†] Manuel E. Acacio[†] José M. García[†] Tim Harris[⋆]
Adrián Cristal[‡◇] Osman Unsal[‡] Ibrahim Hur[‡] Mateo Valero[‡]

Universidad de Murcia[†] Microsoft Research[⋆] Barcelona Supercomputing Center[‡]
IIIA - Artificial Intelligence Research Institute CSIC - Spanish National Research Council[◇]

{rtitos,meacacio,jmgarcia}@ditec.um.es tharris@microsoft.com
{adrian.cristal,osmal.unsal,ibrahim.hur,mateo.valero}@bsc.es

Abstract

In this paper we propose conflict-defined blocks, a programming language construct that allows programmers to change the concept of conflict from one transaction to another, or even throughout the course of the same transaction. Defining conflicts in software makes possible the removal of dependencies which, though not necessary for the correct execution of the transactions, arise as a result of the coarse synchronization style encouraged by TM. Programmers take advantage of their knowledge about the problem and specify through confict-defined blocks what types of dependencies are superfluous in a certain part of the transaction, in order to extract more performance out of coarse-grained transactions without having to write minimally synchronized code. Our experiments with several transactional benchmarks reveal that using software-defined conflicts, the programmer achieves significant reductions in the number of aborted transactions and improve scalability.

TRANSACT '10 April, Paris.

Reprinted from TRANSACT '10, [Unknown Proceedings], April, Paris., pp. 1–8.

1. Introduction and Motivation

Transactional Memory (TM) has been proposed as an easier-to-use programming model that can help developers build scalable shared-memory data structures, relieving them from the burdens imposed by fine-grained locking. By using transactions to synchronize the accesses to shared data, programmers need not reason about interleavings or deadlocks to write correct multithreaded code. Futhermore, they are encouraged to use a coarse-grained style of synchronization, since the underlying system executes critical sections speculatively and can potentially achieve performance comparable to fine-grained locks. Unfortunately, the reality is that large transactions often have data dependencies with other concurrent transactions, and such *conflicts* degrade performance as a consequence of the stalls and/or aborts required to resolve them.

In this context, the use of coarse-grained synchronization becomes a double-edged sword that may create unnecessarily large transactions, introducing artificial conflicts that are unnecessary for correct program execution. In the TM terminology, a conflict is said to occur when two concurrent transactions access the same memory location, and at least one of the accesses is a write operation. According to this low-level definition, hardware transactional memory (HTM) systems typically provide a strict memory consistency model in which all memory references from a transaction that commits earlier seem to happen before all memory references from a transaction that commits afterwards.

However, when data dependencies among transactions are observed from a higher abstraction level, programmers are often able to formulate looser definitions of exactly which conflicts are actually important for a given transaction—e.g., exploiting knowledge about the algorithm, the semantics of the operations, etc. A typical example of this scenario is the *insert* operation in a sorted linked list, which must traverse it until the correct position for the new element is found. A simple thread-safe version of such an operation would simply wrap the sequential code in a transaction (Figure 1a). Consequently, once an insertion begins its execution no other operation can concurrently modify the part of the list that has already been traversed by the insertion. However, taking a closer look at the semantics of insert, a more experienced programmer will realize that this restriction is unnecessary, since there is no harm in allowing other transactional insertions, deletions, etc. to take place on the elements which an insert transaction has left behind.

This observation can be expressed in terms of the way in which conflicts are defined: in this example, write-read dependencies *should not* be considered as conflicts for the insert operation. If this kind of information is provided by the programmer, then the underlying TM system can dynamically adjust the conditions for signalling a conflict, relaxing or tightening them, thus creating opportunities for more parallelism. Yet there exists no programming language construct that allows programmers to change the definition of conflict throughout a transaction, nor the hardware mechanisms to support a variable conflict definition controlled in software.

Furthermore, the implementation of the conflict management mechanism must be efficient, given its critical role in the system. Yet its design should be flexible enough so that TM hardware can be applied toward solving problems beyond guaranteeing mutual exclusion during the execution of critical regions [4]. A well-known example of flexible design is the inclusion of an instruction to explicitly abort a transaction and roll back its tentative work. Programmers find useful the ability of transactions to explicitly rollback execution upon a certain condition, which need not necessarily be a conflict with other transaction. Following the same principle, having a conflict detection hardware whose functionality is configurable in software makes possible to customize or even switch off this component for certain transactions.

```
int TMinsert(List *list,
             string key,
             int data){

  atomic{
    return insert_seq(list,
                      key,
                      data);
  }
}
```

(a) Coarse-grained transaction

```
int TMinsert_notWR(List *list,
                   string key,
                   int data){
  atomic{
    defconflict(!WR){
      return insert_seq(list,
                        key,
                        data);
    }
  }
}
```

(b) Software-defined conflicts

Figure 1. Sample code for the *insert* operation in a linked list.

The first contribution of this paper is the introduction of *conflict-defined blocks*, a new construct which allows TM programmers to eliminate dependencies that are irrelevant to the semantics of their transactions. Figure 1b illustrates the use of these blocks with the list-insertion example. While there are other constructs motivated by performance optimizations, such as early release [11] or open nesting [7–9], their use entails a significant increase in the programming and hardware complexity in comparison to our proposal. We illustrate the use of the proposed construct with examples of its application in well-known transactional benchmarks.

As the second contribution, we present a hardware implementation of this programming construct in a popular HTM system based on LogTM-SE [13]. We show how an HTM system can be augmented with an interface to support software-controlled conflicts, and how this interface can be used by application developers through conflict-defined blocks, or directly by system programmers. For example, programmers can find the rollback functionality useful in algorithms that otherwise need deadlock-recovery mechanisms, but do not require isolation in the execution of such transactions. Other programmers may choose to disable the conflict detection component entirely and instead explicitly introduce their own detection code, for example, to avoid false conflicts provoked by the granularity of the detection. Therefore, our hardware TM system with software-defined conflicts enhances the flexibility of the whole design.

Our third contribution is demonstrating that using software-defined conflicts, programmers can extract more performance from coarsely synchronized code, by taking advantage of the their abstract knowlegde. We augmented three STAMP benchmarks with conflict-defined blocks, and their evaluation on the LogTM-SE system shows how the combination of this programming construct with the appropriate hardware support is able to decrease the number of aborted transactions between 50 and 90% for 16 and 32-thread configurations, and consequently reduce execution time.

The rest of the paper is organized as follows: Section 2 discusses related work. In Section 3 we describe in detail the proposed language construct and hardware support. In Section 4 we discuss some practical examples of application for software-defined conflicts. Section 5 evaluates the performance gains that can be achieved by relaxing the definition of conflict. We end with Section 6, that summarizes the main conclusions of this study.

2. Related Work and Discussion

Several programming language constructs motivated by performance optimization have been proposed for TM. Early release (ER) [11] allows a transaction to remove a data address from its transactional read-set before it commits. This allows other writer transactions to proceed without generating a conflict with the releasing transaction, reducing the probability of long stalls and aborts. However, ER has several important drawbacks that question its usefulness. First of all, programmers must be extremely careful about

```
int FGinsert(List *list,
             string key,
             int data){
  ...
  Lock(list->head->lock);
  while(cur!=NULL){
    Lock(cur->lock);
    if(key<=cur->key){
      ins->next=cur;
      prev->next=ins;
      Unlock(prev->lock);
      Unlock(cur->lock);
      return 1;
    }
    Unlock(prev->lock);
    prev=cur;
    cur=cur->next;
  }
  ins->next=NULL;
  prev->next=ins;
  Unlock(prev->lock);

  return 1;
}
```

(a) Fine-Grained locking (FG)

```
int ERinsert(List *list,
             string key,
             int data){
  ...
  atomic{
    while(cur!=NULL){

      if(key<=cur->key){
        ins->next=cur;
        prev->next=ins;
        Release(&prev->next);
        Release(&ins->next);
        return 1;
      }
      Release(&prev->next);
      prev=cur;
      cur=cur->next;
    }
    ins->next=NULL;
    prev->next=ins;
    Release(&prev->next);
    Release(&ins->next);
    return 1;
  }
}
```

(b) TM + Early-Release (ER)

Figure 2. Sample parallel codes for insertion.

when, where, and with which address ER is used, to avoid violating the overall application atomicity and consistency. Because programmers must explicitly specify *each* memory address to be removed from the read set, ER is very low-level construct whose use entails a difficulty that resembles a lot that of fine-grained locks. As we can observe by comparing Figures 2a and 2b, the programmer needs to include a *release* statement for almost every place where an *unlock* operation appears in the lock-based code. On the hardware side, ER is not easy to support in all HTM systems. For example, systems that use hash signatures do not have the possibility of "removing" an individual address from the signature. Using RW-bits in the private cache makes early release easy to carry out when the block is cached, but it becomes quite tricky when the block has been evicted, since the clean-up of transactional state at the directory takes place lazily, not explicitly. Furthermore, early release is difficult to implement consistently in HTM systems that use cache line granularity, since an early release instruction expects a word address and thus it is not safe to release the entire cache line.

Open nesting [9] is another programming language construct motivated by performance. Open nested transactions can improve concurrency by relaxing the atomicity guarantee. When an open nested transaction commits, the TM system releases its read and written data so that other transactions can access them without generating conflicts. Thanks to open nesting, otherwise-offending transactions can access the exposed data after the nested transaction commits, while the outer transaction still runs. This can enhance the degree of concurrency achieved by a flatenning scheme, which enforces isolation until the outermost transaction commits. When compared to our scheme, open nesting limits available parallelism because full conflict detection is enforced until the nested transaction commits, unlike software-defined conflicts. For instance, continuing with the example presented above, a programmer could wrap the search phase of the insert operation inside a read-only open nested transaction, releasing its entire read set at once when it commits and allowing other insertions to proceed after the insertion point has been found, but not sooner. With software-defined conflicts, however, the system can be configured to not retain isolation over certain transactional data, so that addresses appear to be immediately released from read and/or write set after the access. On the programmer side, open nesting requires commit and abort handlers to be written for each nested transaction. Compensation actions are run when the enclosing transaction aborts, as simply restoring the values of memory locations modified by the nested transaction is insufficient. This additional burden is somewhat similar to that introduced by transactional boosting [3], another technique aimed at enhancing the concurrency of data structures in which the programmer writes inverse methods for recovery, to undo the side effects of an aborted boosted transaction.

On a different matter, the idea of controlling the conflict management hardware in software has been previously explored. FlexTM [10] proposes a set of hardware mechanisms that are software-accessible. This hybrid approach achieves policy flexibility by allowing software to determine when to manage conflicts, either eagerly or lazily. With software-defined conflicts, we add flexibility to the detection stage itself (what is considered a conflict), rather than on the policy (when/how are conflicts handled).

3. Software-Defined Conflicts in Hardware TM

In this section, we describe the proposed language features to support programmer-defined conflicts inside transactions, along with the ISA changes to support this construct in a hardware TM system.

3.1 Conflict-Defined Blocks

Conflict-Defined Blocks (C-def's) replace the fixed concept of "conflict", with a variable definition that is controllable in software. This new construct helps programmers get a tighter grasp on the conflict management mechanism, as they are now able to configure the functionality of such a basic transactional component by simply adjusting the definition of conflict throughout the course of a transaction. This is a powerful yet intuitive feature in the TM programming model that had not been considered in the past.

As a programming language construct, the semantics of C-def blocks is independent from the system in which the program executes. However, the underlying detection policy determines the types of conflicts that can be detected, and thus narrows the spectrum of conflict types on which the definition is based upon. Hence, the variety of optimizations offered by C-def blocks is restricted by when and how conflicts are detected, as well as by the data versioning policy used. In this way, systems that defer the detection until transaction commit (a.k.a. *lazy* or *optimistic*), carry out the detection by communicating the write set of the committer transaction to the rest. Therefore, a conflict for a lazy transaction means that it read (and/or wrote) data too early, because a remote transaction that "happens before" this one has changed it. On the other hand, a conflict in eager policy means that a remote transaction *wants* to write (read) data that this one has already read/written (written). In this paper, we explore the applications of software-defined conflicts in a TM system that employs eager conflict detection and eager version management. The use of conflict-defined blocks on eager-lazy and lazy-lazy systems is left as future work.

Dependence Types and Definition of Conflict. A transaction's definition of conflict is a *local* concept given by its response to each of the four possible kinds of interactions or dependences that the *local* transaction can experience with other *remote* concurrent transaction: read-write (RW) –a remote transaction wants to read data locally written–, write-read (WR), write-write (WW) and read-read (RR). The definition controls which kinds of dependences are signaled as conflict, but it does not say anything about the resolution process (i.e. which transaction must be aborted). In an eager system, the coherence protocol ensures that a transaction observes those requests issued by remote transactions that are potentially offending (i.e. for lines that may be in its read and write sets). The transaction then uses its local definition of conflict to decide whether the observed access must be considered conflicting or, on

the contrary, entails a dependence whose type the programmer has explicitely allowed using conflict-defined blocks. The traditional definition of conflict, which we refer to as the *default definition* throughout this paper, considers conflicting those types of dependencies in which at least one of the two transactions writes the data (WW, RW, WR). Moreover, we use the term empty definition to refer to that definition in which none of the dependence types are considered to be conflicting.

Syntax. The syntax of this construct comprises the `defconflict` keyword, the types of conflicts to add/remove from the current definition, and finally the block (statements and declarations) for which the new definition of conflict applies.

```
defconflict  ( <dep_types> )
{
     BLOCK
}

<dep_types> := <dep_type> [, <dep_types>]

<dep_type> := [!] {WW|RW|WR|RR|ALL|NONE}
```

Not every dependence type needs to be specified in each C-def block, but only those the programmer wishes to redefine, if any. Each type can only appear once in the dependence type list. Each type may be optionally negated. `ALL` is equivalent to the default definition `(RW,WR,WW)`, while `NONE` is synonymous with `!ALL`, representing an empty definition in which no conflicts are detected while executing the C-def code block. The block is a regular section of code grouped together, consisting declarations and statements. Multiple C-def blocks can be nested.

Semantics and Scope. With the introduction of software-defined conflicts, a new concept of *current definition of conflict* comes into scene. The current definition appears as an implicit, thread-local variable with dynamic-scope, which determines whether the thread considers other memory accesses as offending. When the program starts, the definition is automatically set to `NONE` so that conflict detection is turned off when executing non-transactional code. New threads inherit the definition of the parent thread at the time they are created, commonly the empty definition. When the thread enters a transaction, the current definition is immediately changed to `ALL`, in order to provide the default definition of "two memory accesses, at least one of them a write". We can think of it as if transactions were implicitly wrapped in the C-def block `defconflict(RW,WR,WW)`. A C-def block then updates the current definition before executing the statements in its block, adding or removing types of dependences. If a type appears negated in the list, it is *disabled* –removed from the definition– and not consider conflicting within the block. If it appears not negated, then that type is *enabled* –added to the definition–. Conflict types that are not specified for a given C-def block remain in the same state as they were outside that block. Adding (removing) a conflict type that was already enabled (disabled) has no effect on the definition.

The programmer perceives the current conflict definition as a variable with dynamic scope. All nested function calls inside the C-def block inherit the same definition, until the program leaves the C-def block or another nested C-def block redefines it. In the latter case, the inner conflict type list overrides the outer definition for the specified types (the types not listed in the inner C-def declaration remain unchanged). When the execution abandons a C-def block the outer definition is restored.

Granularity. The current definition is applied to *all* incoming data requests while inside the C-def block, to determine whether they are conflicting or not. Therefore, C-def blocks are a coarse-grained mechanism of releasing/reacquiring isolation in regards to addresses, unlike early release. However, from a coding perspective, the programmer is free to use C-def blocks with a finer or coarser block granularity, from wrapping the entire transaction to having only a few statements inside a C-def block.

Dependence Transformation. C-def blocks guarantee that, once a remote transaction has trespassed a local transaction's R&W sets, the latter cannot access such data again throughout the course of its execution, and any attempt to do so will cause its immediate abort. For instance, when WR dependences are disabled, the local transaction is not allowed to reload nor modify any Rset data that has been written by a remote transaction. If RW conflicts are disabled, the local transaction is not allowed to modify any Wset data that has been read by a remote transaction.

3.1.1 Relaxed Transactional Consistency Models

Generally, HTM systems assume a fixed definition of conflict that leads to a simple, straight-forward model of sequential consistency (SC) based on transactions, a model that is easy for the programmer to reason about.

Transactional SC is a strict memory consistency model that imposes strong limitations on the possible interleavings of transactions, thus limiting concurrency and performance. However, those severe restrictions are not required in all transactions, and some codes can still produce correct results despite relaxing the definition of conflict. Such changes of the definition are accomplished through C-def blocks, introducing relaxed transactional memory consistency models.

Shared memory systems have likewise implemented relaxed memory consistency models that allow for further optimizations in comparison to SC. In this context, we believe it makes sense to offer the TM programmer the possibility to trade some of the programming ease for performance, by using a relaxed transactional memory consistency model. Just like these models allow for reorderings of memory operations that are illegal under SC, C-def blocks permit overlappings in the read and write sets that are not possible under transactional SC. Of course, these forbidden interleavings make it more difficult to reason about the correctness of a program. Therefore, C-def blocks are intended to be used by more experienced programmers who are willing to address the added complexity in their quest for performance.

Stale Consistency. When a transaction T_1 disables write-read (WR) conflicts, it allows other transaction T_2 to write on blocks that T_1 has read, and both continue their execution as if no conflict had happened. If the offender T_2 commits while T_1 is still running, the values read by T_1 become *stale* when T_2 publishes its updates –hence its name–. Nothing stops T_1 from observing T_2's updates after T_2 has committed, except for those lines that are also part of T_1's read set (since that entails a forbidden dependence transformation). Therefore, the atomicity property is not preserved, as T_1 may read both pre- and post-T_2 values throughout its execution. On the other hand, the execution of T_1 does appear atomic when it commits before any offending writer, and in this case the result of the execution is equivalent to the transactional SC model.

Producer Consistency. When a transaction T_1 disables its read-write (RW) conflicts, it allows other transactions T_2, T_3 to read blocks that T_1 has written. The consumers then continue their execution as if no conflict had happened and may commit even before the producer. However, if T_1 eventually aborts, the conflicted data is restored without further notification to the consumers.

3.2 Hardware and Compiler Support

Adding support for software-defined conflicts requires simple changes at both the compiler and hardware levels. On the one hand,

the hardware must provide an interface for setting and obtaining the definition of conflict in software. Many previous HTM proposals incorporate a *transaction status register* visible in software, which we simply augment with three new bits to encode the current definition of conflict. Each type of dependence (WR,WW,RW) is represented by one bit in the register, which indicates whether that kind is enabled (consider conflicting) in that moment. The conflict check logic is slightly modified so that these three bits control how the output from the query of the read and write sets must be interpreted depending on the type of coherence request received –unlike traditional HTM systems, which embed this decision in silicon–. The hardware is extended with a *conflict signature* that summarizes the addresses of those remote accesses that would have caused a conflict, had not it been for the relaxed definition established by a C-def block. After an address is added to this conflict signature, it must be checked against every local load or store until the end of the transaction, in order to avoid dependence transformation. Finally, for eager-versioned systems, the logging logic must be made configurable and capable of operating with word granularity as well as lines.

The bits that hold the conflict definition in the transaction status register are manipulated through regular ISA instructions that the compiler inserts in the appropriate places of the code. At the beginning of each C-def block, the compiler must place some instructions to load the transaction status register and save the current definition of conflict in a space allocated in the local stack. Then, the corresponding bits of the status register are set or cleared according to the list of dependences types given in the C-def block. When the end of the C-def block is reached or it is abandoned through a *break*, *return* or other similar statement, the previous definition is recovered from the stack and restored into the status register. Finally, the compiler assumes that each regular transaction has an implicit C-def block with the default definition of conflict.

4. Applications and Use Cases of C-def Blocks

In this section, we show some practical examples of how C-def blocks can be used in real applications to reduce the amount of conflicts suffered by the transactions.

4.1 Emulating Early Release: *genome* and *labyrinth*

C-def blocks are useful to remove unessential dependencies that commonly occur when traversing data structures in search of some element. For example, two of the transactions in the benchmark genome of the STAMP suite [2] comprise a single insert operation into a sorted linked list that saves the hashes of segment substrings. We wrap the body of each transaction in a C-def block that undefines WAR conflicts, as shown in Figure 1, to allow other insertions that modify the elements which another concurrent transaction has traversed while searching for the location of the insertion.

The labyrinth benchmark is another example of algorithm in which the programmer can attempt to extract additional performance by adjusting the definition of conflict to the semantics of the problem. Labyrinth implements a variant of Lee's algorithm [12]: For each pair of input points, the program finds the shortest route that connects them in a three-dimensional uniform grid that represents the maze. The main transaction of the program encloses the calculation of the path and its addition to the global grid. To avoid unnecessary writes to the global grid, each thread creates a local copy of the global grid and uses it for the route calculation (expansion and trace-back phases). The pseudocode of the main transaction can be observed in Figure 3.

At a high abstraction level, a conflict occurs when two threads pick paths that overlap. This can be expressed in terms of the conflict definition as "only write-write conflicts on the global grid are meaningful". However, before the introduction of software-defined conflicts, the programmer had no way to provide such valuable information to the underlying levels. In the following paragraphs, we discuss how C-def blocks can be used to properly adjust the conflict definition to eliminate unwanted dependencies without adding any complexity to the reasoning process, and with minimal changes to the original code.

```
Grid global;

forall routes {
        atomic {
            defconflict(!WR) {
                Grid local;

                Copy global grid into local grid;
                Expand from source to destination; // in local
                Traceback from destination to source; // in local
                Add found path to global grid;
            }
        }
}

void grid_addPath (...) {
    forall points in path {
         if point is full in global grid then abort transaction
         else mark point as full in global grid
    }
}
```

Figure 3. Pseudo-code for the labyrinth program with software-defined conflicts

Default Conflict Definition. In the process of creating a private copy of the grid, transactions add the entire global grid to their read sets, causing write-read conflicts whenever one of them attempts to add its calculated path to the global grid. Obviously, this definition completely serializes transactions because, no matter how many concurrent transactions are calculating paths in a given moment, only one of them will be able to update the global grid, while the rest will have to abort. If the TM system supports it, the benchmark can use early release in order to remove the global grid from the transaction's read set, after the grid copy has completed. Releasing addresses from the read set requires the grid points along the new path to be revalidated when the path is added, as we can see in the *addPath* function of Figure 3. This is to make sure that none of the selected points has become part of another path after the grid copy.

Disabling Write-Read Conflicts. By using C-def blocks, a programmer can eliminate WR conflicts without having to manually release addresses and then worry about validating the found path. The changes to the original code are straight-forward, and consist of wrapping the main transaction in a C-def block. Under this definition, transactions can add paths to the global grid while other transactions are calculating their route. Using the relaxed definition for all the transaction opens more opportunities for concurrency at the cost of risking an inconsistent snapshot, because paths would not appear to be added atomically (T_1's copy could observe only part of a path committed by T_2 during its copy). Nonetheless, correctness is never threatened since any attempt to update a global point that has been added to another path since the grid copy, entails a forbidden dependence transformation that will force the transaction to abort, even if paths are not revalidated at the end.

4.2 Avoiding False Conflicts: *labyrinth*

If the underlying TM system tracks conflicts on a cache-line granularity –as most HTM systems do–, unnecessary conflicts arise in the benchmark labyrinth as a result of false sharing amongst points of the same row. For example, when a grid of size 32 is mapped to hardware that uses 64-byte cache lines, each row of the grid occupies only two lines (16 points per line, 4 bytes per point). When a

transaction adds one point to its path, concurrent attempts to mark the row-adjacent points of the line will create a false write-write conflict, causing unnecessary stalls or aborts. If the hardware employs lazy versioning (or it supports eager versioning with word granularity), C-def blocks can be used to fight false conflicts.

Disabling Conflict Detection. Thanks to software-defined conflicts, experienced programmers also have a tool to address false sharing in their applications by assuming full responsibility on the conflict detection. Wrapping the main transaction of labyrinth in the C-def block `defconflict(NONE)` disables all conflicts types, so that transactions are exclusively used for its ability to rollback. Indeed, the "undo" capability of transactions is actually what makes them beneficial for implementing this program, as deadlock avoidance techniques would be required in a lock-based approach. Detecting conflicts manually in labyrinth is so straight-forward that is in fact already implemented in the benchmark, in order to support early release. The program *explicitly* checks for conflicts when it revalidates (re-reads) the points along the found path before adding it to the global grid, to make sure that none of them has been marked as full by other concurrent transaction. If a full point is found along the path, a programmer-triggered abort is issued; otherwise, the point is marked as full. To ensure correctness, this check-and-update must be performed atomically, either by using a compare-and-swap instruction, or even by using the C-def block `defconflict(ALL)` inside the loop of the addPath function.

4.3 Avoiding Starving-Writer Problems: *vacation*

The vacation benchmark can also benefit from a relaxed conflict definition. This application implements an on-line transaction processing system similar in design to SPECjbb2000. The database is implemented as a set of trees that keep track of customers and their reservations for various travel items. Client threads perform a number of sessions that interact with database in three types of sessions: reservations, cancellations, and updates. Each session is enclosed in a coarse-grain transaction to ensure validity of the database. The main transaction of vacation, as shown in Figure 4, has two phases: First, the items solicited by the client are searched in the database. Then, if the queries returned some matching results, a reservation is made each matching type of item.

When this benchmark is executed in a system with eager conflict detection, those transactions that attempt to update the database tables by adding or removing new items (car, flights, etc.), may suffer a pathological interaction commonly referred as *starving writer* [1]: They may find very difficult to acquire exclusive ownership over those cache lines that are heavily accessed by other reader transactions that only query the table. Even if the priority scheme dictates the abort of any concurrent query, by the time its rollback completes other query could have loaded the conflicting line, impeding any forward progress on the writer side.

```
Grid global;

void client_run(...)
...
        atomic {
          defconflict(!WR) {
            for item in requested items do {
                query database and record matches
            }
          }
          if matched then make reservation
        }
    }
}
```

Figure 4. Pseudo-code of the main transaction in vacation with software defined conflicts.

Table 1. System parameters.

MESI Directory-based CMP	
Core Settings	
Cores	2 to 32, single issue in-order, non-memory IPC=1
Memory and Directory Settings	
L1 I&D caches	Private, 32KB, split 2-way, 1-cycle latency
L2 cache	Shared, 512KB per tile, unified 4-way, 12 cycle-latency
L2 Directory	Bit vector, 6-cycle latency
Memory	4GB, 300-cycle latency
Network Settings	
Topology	2D Mesh (4x4)
Link latency	1 cycle
Link bandwidth	40 bytes/cycle

Disabling write-read conflicts. Using C-def blocks as depicted in Figure 4, a programmer allows writer transactions to proceed despite the presence of concurrent readers (queries), and resolve the problem of starving writer. The effect of relaxing the definition in this way is that the state of the database may change from the query phase to the reservation phase. However, a transaction never observes the database in an inconsistent state, because C-def blocks avoid such transformations of conflicts: If in its attempt to make a reservation, a transaction reaccesses data that has changed since the query phase, it will be forced to abort.

5. Evaluation

In this section, we evaluate the performance of the proposed programming language construct using an HTM that supports software-defined conflict detection.

5.1 Simulation Environment

We use a full-system execution-driven simulator based on the Wisconsin GEMS tool-set [6], in conjunction with Virtutech Simics [5]. We use the implementation of LogTM-SE [13] and the detailed timing model for the memory subsystem of GEMS v2.1, with the Simics in-order processor model. Simics provides functional correctness for the SPARC ISA and boots an unmodified Solaris 10. LogTM-SE extends a directory protocol to perform eager conflict detection, and encodes read and write sets using hash signatures. LogTM resolves conflicts through stalls, and uses a deadlock avoidance algorithm based on timestamps. We use an ideal book-keeping scheme to track read and write sets (*perfect signatures*).

We perform our experiments on a tiled CMP system, as described in Figure 4.3. We use 2 to 32-core configurations with private L1I and L1D caches and a shared, multibanked L2 cache of 512KB (one L2 slice per tile). The L1 caches maintain inclusion with the L2. The cores and L2 cache banks are connected through a 2D mesh network. The private L1 data caches are kept coherent through an on-chip directory (at L2 cache banks), which maintains a bit vector of sharers and implements the MESI protocol. We use three benchmarks extracted from the STAMP suite [2]: labyrinth, vacation-high and genome. We use the small input sets for the two former applications, and the medium size input for the latter. To minimize the influence of the operating system and avoid pathological interactions between OS and the transactional application, we leave one core idle in each experiment and we run n-1 threads for the experiments in an n-core configuration. We also disable interrupts for the remaining cores and bind each thread to one core.

5.2 Results

The resuls of the evaluation are summarized in Figure 5.2, which shows the speedup (left side) and the number of aborted trans-

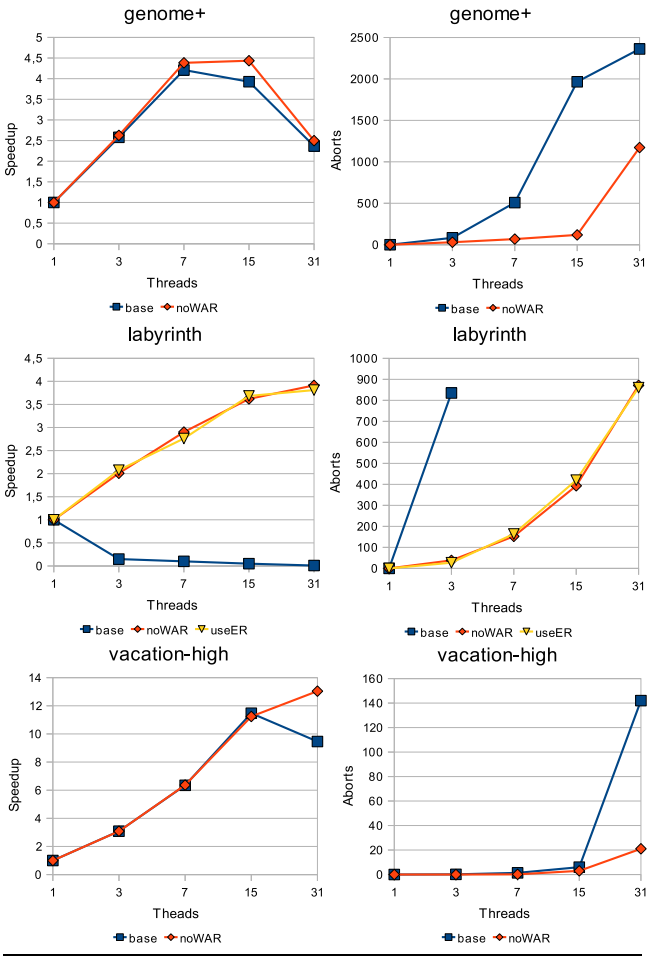

Figure 5. Speedup and number of aborted transactions.

actions (right side) for each of the three considered benchmarks. We compare the performance of the baseline LogTM-SE system (*base* plot in the graphs) against the same HTM system extended to support software-defined conflicts (*noWAR* plot). For the considered benchmarks, the definition was relaxed by disabling write-read conflicts as discussed in Section 4.

For genome, disabling write-read conflicts for the two transactions that perform insertions in a list completely eliminates the aborts suffered by these two transactions. A significant portion of the total number of aborts suffered for configurations of 7 or more threads correspond to one of these transactions. These aborts are caused by concurrent insertions in the same table (dynamic instances of the same static transaction competing for the list). As mentioned earlier, relaxing the definition for this list insert operation acts as a straight-forward implementation of early release, achieved by simply adding a single line of code. For 15 threads, the number of aborts is reduced from almost 2000 in the baseline system (*base* plot) to less than 100 (*noWAR* plot). The effect on the execution time is not as impressive (since this phase of insertions only spans a small amount of the total execution time), though we can see how it improves the scalalabilty (up to 15 threads). For larger configurations, the contention experienced by other "not relaxed" transactions increases the number of aborts and degrades the overall performance in both cases.

Regarding labyrinth, as explained in Section 4, the base system (without ER nor C-def blocks) performs very poorly, as only one of the transactions is able to update the global grid at a time. Even worse, with configurations of 7 or more threads, the benchmark struggles to make forward progress and the execution time skyrockets, because the contention for the global grid is so high that the highest priority transaction cannot add its path until all other threads have aborted several times and are backing off. This causes the starvation of the writer for long periods and quasi-livelock scenario. By disabling WAR conflicts, on the other hand, transactions are allowed to add paths to the grid while other transactions keep calculating their route, which results in a scalable behaviour up to 31 threads and a speedup of 4 (*noWAR* plot). To provide a fair comparison, we also evaluate an ideal implementation of early release (*useER* plot), since the benchmark is designed to use this programming construct. Note that early release is simulated by removing line addresses from the perfect signature that represents the transaction's read set, and could not be feasible if true hash signatures were employed. We can observe in Figure 5.2 that using C-def blocks to disable WAR conflicts emulates the functionality of an ideal early release (*useER* plot), in both number of aborts and execution time.

As for vacation-high, we observe how the performance scales very well up until 15 threads for both configurations, because of a very low or inexistent number of aborts. For the 31-thread experiment, the number aborts in the base case goes up by a factor of 20, and this is reflected on a performance degradation that leads to worse execution time than for 15 threads. The aborts mostly occur when a transaction attempts to add or remove items from the database (several red-black trees). The situation is aggravated by the starving writer pathology: The "updater" needs to perform some updates (adjust colours, rebalance, etc.) in a data structure which is heavily accessed by many reader transactions (queries). When the updater becomes the highest priority, all cyclic conflicts result in the abort of the readers, hence the sudden increase in the number of aborts (the vast majority are reservation queries). However, their constant retries not only are fruitless but also prevent the writer from acquiring exclusive ownership and write the required cache lines, and this pathological interaction only ends when their backoff period grows long enough to give way to the writer. On the other hand, the version with software-defined conflicts manages to keep the aborts within reasonable bound and allows the benchmark to scale up to 31 threads. The temporary relaxation of write-read conflicts in the noWAR case allows the writer to proceed in the presence of readers, solving the starving writer scenario, and dramatically reduces the number of aborts.

6. Conclusions

In this paper, we have given the concept of conflict definition an interesting twist, transforming it from its fixed nature to a variable, software-controllable definition. We have introduced conflict-defined blocks, a new programming construct that enables the programmer to alter the definition throughout the execution of a transaction. C-def blocks are used to relax the conditions for signalling a conflict in those cases where relaxing the isolation guarantees does not threaten correctness, in order to eliminate conflicts that are not essential to the execution.

We have presented several examples of its application on common scenarios and well-known transactional benchmarks. We have described a straightforward implementation of this programming construct on top of an eager HTM system. Finally, we have evaluated the proposed hardware/software scheme on top of a popular HTM system such as LogTM-SE using real applications, and we have demonstrated that software-defined conflicts can significantly reduce the number of aborts in the selected applications, obtaining more performance out of coarse-grained transactions.

References

[1] Jayaram Bobba, Kevin E. Moore, Luke Yen, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional memory. In *Proceedings of the 34rd Annual International Symposium on Computer Architecture*, pages 81–91, Jun 2007.

[2] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of The IEEE International Symposium on Workload Characterization*, pages 35–46. Sept 2008.

[3] Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, feb 2008.

[4] Mark D. Hill, Derek Hower, Kevin E. Moore, Michael M. Swift, Haris Volos, and David A. Wood. A case for deconstructing hardware transactional memory systems. Technical report, Univ. of Wisconsin Computer Sciences Technical Report CS-TR-2007-1594, 2007.

[5] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb 2002.

[6] Milo M.K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *Computer Architecture News*, pages 92–99, September 2005.

[7] Austen McDonald, JaeWoong Chung, D. Carlstrom Brian, Chi Cao Minh, Hassan Chafi, Christos Kozyrakis, and Kunle Olukotun. Architectural semantics for practical transactional memory. pages 53–65, Jun 2006.

[8] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting nested transactional memory in LogTM. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 359–370, Oct 2006.

[9] Yang Ni, Vijay S. Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *Proceedings 12th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 68–78, mar 2007.

[10] Arrvindh Shriraman, Sandhya Dwarkadas, and Michael L. Scott. Flexible decoupled transactional memory support. In *Proceedings of the 35th International Symposium on Computer Architecture*. Jun 2008.

[11] Travis Skare and Christos Kozyrakis. Early release: Friend or foe? In *Workshop on Transactional Memory Workloads*. Jun 2006.

[12] Ian Watson, Chris Kirkham, and Mikel Lujan. A study of a transactional parallel routing algorithm. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 388–398, Washington, DC, USA, 2007. IEEE Computer Society.

[13] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, pages 261–272, Feb 2007.