Hardware Transactional Memory with Software-Defined Conflicts

RUBEN TITOS-GIL, MANUEL E. ACACIO, and JOSE M. GARCIA,
Universidad de Murcia
TIM HARRIS, Microsoft Research
ADRIAN CRISTAL, OSMAN UNSAL, IBRAHIM HUR, and MATEO VALERO,
Barcelona Supercomputing Center

In this paper we investigate the benefits of turning the concept of transactional conflict from its traditionally fixed definition into a variable one that can be dynamically controlled in software. We propose the extension of the *atomic* language construct with an attribute that specifies the definition of conflict, so that programmers can write code which adjusts what kinds of conflicts are to be detected, relaxing or tightening the conditions according to the forms of interference that can be tolerated by a particular algorithm. Using this performance-motivated construct, specific conflict information can be associated with portions of code, as each transaction is provided with a local definition that applies while it executes. We find that defining conflicts in software makes possible the removal of dependencies which arise as a result of the coarse synchronization style encouraged by the TM programming model. We illustrate the use of the proposed construct in a variety of use cases with real applications, showing how programmers can take advantage of their knowledge about the problem and other global information not available at run-time. We describe how to implement a hardware TM design that utilizes this software construct. Our experiments reveal that leveraging software-defined conflicts, the programmer is able to achieve significant reductions in the number of aborts –over 50% for most applications. At 16 threads, our system with software-defined conflicts outperforms LogTM-SE in nearly all benchmarks, reaching an average reduction in execution time of 18%.

Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors); D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms: Design, Performance

Additional Key Words and Phrases: Hardware transactional memory, conflict detection, atomic block

ACM Reference Format:
Titos-Gil, R., Acacio, M. E., Garcia, J. M., Harris, T., Cristal, A., Unsal, O., Hur, I., and Valero, M. 2012. Hardware transactional memory with software-defined conflicts. ACM Trans. Architec. Code Optim. 8, 4, Article 31 (January 2012), 20 pages.
DOI = 10.1145/2086696.2086710 http://doi.acm.org/10.1145/2086696.2086710

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

This work was supported by the Spanish MEC and MICINN, as well as European Comission FEDER funds, under grants Consolider Ingenio-2010 CSD2006-00046 and TIN2009-14 475-C04. R. Titos-Gil has a research grant from the Spanish MEC under the FPU National Plan (AP2006-04152).
Authors' addresses: R. Titus-Gil, M. E. Acacio, and J. M. Garcia, Universidad de Murcia; T. Harris, Microsoft Research; A. Cristal, O. Unsal, I. Hur, and M. Valero, Barcelona Supercomputing Center. Correspondence email: rtitos@ditec.um.es.

DOI 10.1145/2086696.2086710 http://doi.acm.org/10.1145/2086696.2086710

```
int TMinsert(List *list,
             string key,
             int data){
  atomic{
    return insert(list,
                  key,
                  data);
  }
}
```

Fig. 1. Coarse transaction.

to synchronize the accesses to shared data, programmers need not reason about interleavings or deadlocks to write correct multithreaded code. Furthermore, TM encourages coarse-grained 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 often creates large transactions that introduce artificial conflicts, not required 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 the references from a transaction that commits afterwards [Ananian et al. 2005; Hammond et al. 2004; Moore et al. 2006].

However, when data dependencies among transactions are observed from a higher abstraction level, programmers are able to formulate looser definitions of exactly which conflicts are actually important for a given transaction—e.g., exploiting knowledge about the algorithm, etc. Indeed, many of the conflicts at the level of memory words are not essential to the high-level semantics of the operations [Ni et al. 2007]. Typical examples of this scenario are the *find, insert* or *delete* operations in a sorted linked list, which traverse it looking for a given element or the correct position where a new one must be inserted. A simple thread-safe version of such operations would simply wrap the sequential code in a transaction (Figure 1). Consequently, once the search phase of the transaction begins, no other list operation can concurrently modify the part of the list that has already been traversed by the former. 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 insertions to operate on the elements that another transaction has already traversed.

This observation can be equivalently expressed in terms of the low-level definition of conflict: in the examples of the sorted-linked list, write-after-read dependencies *should not* be considered as conflicts. 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 constructs that allow programmers to change the conflict definition for a transaction, nor the hardware mechanisms and interfaces to support such variable concept of conflict controlled in software.

The implementation of the conflict management mechanism must be efficient, given its critical role in the system, while its design should be flexible enough so that TM hardware can be applied towards solving problems beyond guaranteeing mutual exclusion during the execution of critical regions [Hill et al. 2007]. A well-known example

```
int TMinsert(List *list,
             string key,
             int data){
  atomic(!WAR){
   return insert(list,
                 key,
                 data);
  }
}
```

Fig. 2. Using PABs.

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.

The first contribution of this paper is the introduction of *parameterized atomic blocks*, a new language construct which allows TM programmers to associate specific conflict information with atomic blocks, in order to eliminate dependencies that are irrelevant to the semantics of those transactions. Figure 2 illustrates the use of these blocks with the list-insertion example. While there are other constructs motivated by performance optimizations, such as early release [Skare and Kozyrakis 2006] or open nesting [Moravan et al. 2006; McDonald et al. 2006; Ni et al. 2007], their use entails a significant increase in the programming and hardware complexity in comparison to our proposal.

As the second contribution, we propose the use of parameterized atomic blocks (from here on, PABs) to enforce transactional execution in mutual exclusion, providing the programmer with a tool to configure the concurrency control scheme employed by the underlying system for given transactions. Thanks to PABs, the programmer can indicate when transactions shall be executed under a pessimistic approach. We illustrate the use of the proposed construct to achieve this lock-like behavior with several examples of its application in transactional benchmarks.

Our third contribution is a hardware implementation of this programming construct in a popular HTM system based on LogTM-SE [Yen et al. 2007]. 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 parameterized atomic 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 design.

Our fourth contribution is demonstrating that using software-defined conflicts, programmers can extract more performance from coarsely synchronized code, by taking advantage of their abstract knowledge. We introduced parameterized atomic blocks into six STAMP benchmarks, raytrace from SPLASH-2, and a parallel implementation of Kohonen's self-organizing maps that we have developed using coarse-grained transactions. The 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.

```
foreach gene in segment length {
  compute start hash of segment
  atomic(!WAR) {
    list_insert(startHash,...)
    }
}
```

Fig. 3. PABs in *genome*.

The rest of the paper is organized as follows: In Section 2 we discuss some practical examples of application for software-defined conflicts. Section 3 describes in detail the proposed language construct, and Section 4 its hardware and compiler support. In Section 5, we evaluate the performance gains that can be achieved by relaxing the definition of conflict. Section 6 discusses related work. We end with Section 7, that summarizes the main conclusions of this study.

2. APPLICATIONS AND USE CASES

In order to enrich the semantics of atomic blocks, we propose the parameterization of this construct as a means to optionally provide additional information to the underlying levels of the TM system, about different aspects of the transaction. To support software-controlled conflicts, we augment such parameter list with an attribute that specifies the definition of conflict for that block. In this way, we propose the *Parameterized Atomic Block* (PABs) as an extension to common atomic blocks, which makes them capable of replacing the fixed concept of "conflict" with a definition that is variable, controllable in software. This extended construct helps programmers get a tighter grasp on the conflict management mechanism, as they are now able to configure its functionality by simply adjusting the definition at the beginning of a transaction. This is a powerful yet intuitive feature in the TM model that had not been considered in the past. In this section, we show some practical examples of how PABs can be used in real applications to reduce the amount of aborts suffered by the transactions. We explain in detail the syntax and semantics of PABs in Section 3.

2.1. Emulating Early Release

Programmers that use coarse-grained transactions to synchronize their code often create atomic blocks that take a snapshot of a shared data object, then perform some computations, and finally update some part of the data structure. In many cases, the loaded data no longer needs to be monitored for conflicts after it has been used, and other transactions can safely trespass the read set of an active transaction without compromising correctness, with the only constraint that the affected transaction does not eventually attempt to write the conflicted data. Thus, the default definition of conflict creates unessential write-after-read (WAR) conflicts that degrade performance. On the other hand, by making use of PABs, programmers can relax such definition, indicating to the underlying TM system that WAR dependencies are not meaningful.

Genome. PABs are useful to remove WAR dependencies that commonly occur when traversing data structures in search of some element. For example, two of the transactions in the benchmark genome from the STAMP suite [Cao Minh et al. 2008] comprise this kind of insert operation into a list. We can redefine the default definition of those coarse-grained transactions and disable WAR conflicts, as shown in Figure 3, in order to allow concurrent insertions on different positions of the list, This relaxed definition achieves similar concurrency as the early release construct, but the latter comes at a complexity much like fine-grained locking [Skare and Kozyrakis 2006].

Labyrinth. The STAMP benchmark labyrinth is another example where PABs are applicable. Labyrinth implements a variant of Lee's algorithm [Watson et al. 2007].

```
forall routes {
    atomic(!WAR) {
        Copy global grid into local grid;
        Expand local from src to dest;
        Traceback local from dest to src;
        Add found path to global grid;
   }
}
void grid_addPath (...) {
    forall points in path: if point full in global
         then abort transaction
         else mark point as full in global
    }
}
```

Fig. 4. Software-defined conflicts in *labyrinth*.

```
forall input samples {
   atomic(!WAR) {
      winner=getWinner(map,inputSample,...);
      neighbourhood=getNeighbours(winner,...);
      updateUnits(winner,neighbourhood,...)
    }
}

void updateUnits(winner,neighbourhood,...) {
  if winner is no longer BMU  then abort
  else update weights of neighbours
}
```

Fig. 5. Software-defined conflicts in *som*.

Its main transaction encloses the calculation of the shortest path between two points and its addition to the global grid. The pseudo-code is shown in Figure 4. 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). As all running transactions add the global grid to their read set in the copy, any attempt to update the global grid with a new path causes the rollback of all other transactions. If the underlying TM system supports early release [Skare and Kozyrakis 2006], the global grid can be removed from the read set after the copy, but then the found path needs to be validated when added, to ensure its points have not become part of another path in the meantime. At a high abstraction level, a conflict only occurs when two threads pick paths that overlap, or expressed in terms of the conflict definition, "only write-write conflicts on the global grid are meaningful". By means of software-defined conflicts, a programmer can eliminate WAR dependencies from labyrinth without having to manually release addresses and then validate the found path. The changes to the original code are straight-forward, as we can see in Figure 4.

Kohonen's Self-Organizing Maps. To show the applicability of software-defined conflicts, we have developed a parallel implementation of Kohonen's self-organizing maps [Kohonen et al. 2001] using transactions (*som*). The training of the map is done in two phases, the ordering phase (coarse-grained adjustment) followed by the convergence phase (fine tuning). There exist a substantial amount of parallelism in the convergence phase, as multiple training samples can be processed in parallel as long as they map to different winner neurons and their neighbourhoods do not overlap. We parallelized this fine-tuning phase using a single coarse-grained transaction to synchronize all accesses to the map, as shown in Figure 5. The result is a parallel program very similar to its sequential counterpart. The transaction encloses both search and update steps, guaranteeing that the best matching unit found stays as winner neuron until the update is complete. Like in labyrinth, the entire map is added to the read

set during the search step, making it impossible for other transactions to update the matrix without generating a conflict under the default definition. However, we can relax the definition using PABs to remove WAR dependencies that are not needed for correctness, and increase concurrency.

2.2. Pessimistic Concurrency Control (PCC)

Transactions are used as a simple, intuitive synchronization abstraction that ensures a critical section is executed atomically and in isolation. How the underlying system guarantees those properties while providing little performance overhead is something that does not concern the common programmer. However, more experienced developers may want to have some degree of control on how the TM system handles concurrency, since using the optimistic approach may not always be the best choice in terms of performance [Sonmez et al. 2009]. In many cases, transactions enclose fine-grained critical sections consisting of read-modify-write operations over a single variable, (i.e. a shared counter, etc.), which are known not to expose parallelism. In spite of that, TM programmers have no way to communicate this particular features to the lower levels, and thus all transactions are executed in the same way and without distinction. Furthermore, mixing lock variables and transactions in order to provide mutual exclusion with pessimistic concurrency control (PCC) introduces many potential problems [Volos et al. 2008], jeopardizes code composability, etc.

PABs also offer the programmer the ability to decide whether a transaction is executed optimistically or pessimistically, contributing to the flexibility of the TM design. Considering read-after-read (RAR) dependencies as conflicting, the programmer can choose to serialize the dynamic instances of a static transaction a priori, even though that kind of interaction does not involve a real conflict. All elements accessed by the transaction are automatically locked and cannot become part of another transaction's read or write set, thus achieving the semantics of mutual exclusion of a lock, for read-only data too. Unlike with single global lock semantics, only the transactionally accessed elements are locked, allowing other concurrent transactions with different data requirements to execute and commit as usual.

Kmeans. The STAMP benchmark kmeans is a clear example of a program in which the programmer does not use transactions for its ability to exploit parallelism out of coarse-grained transactions. Transactions are used to protect the update of the cluster centre that occurs during each iteration. The majority of execution time for kmeans is spent calculating the new cluster centres [Cao Minh et al. 2008], an operation that threads performs locally without the need for synchronization. K-means benefits from optimistic concurrency if threads update different centres concurrently, but it may turn counter-productive as contention grows, since the imprecise nature of back-off strategies can leave threads stalling for too long, after a quick burst of aborts. Enabling RAR conflicts to serialize transactional execution may be helpful when contention is high.

Raytrace. The benchmark raytrace from the SPLASH-2 suite [Woo et al. 1995] is another example of a parallel program in which the work is divided amongst threads that perform most of the computations locally, and only need to synchronize when accessing to a global ray identifier. Transactions are used uniquely to ensure exclusive access to that variable. Aside of other drawbacks of locking, transactions may entail an improvement over locks because they avoid the overhead of the lock acquire and release operations, which becomes a considerable bottleneck in a program with hundreds of thousands of tiny transactions.

Yada, Intruder. In yada, the programmer utilizes a transaction to protect the access to a work-list (a heap) from which threads extract elements for processing. Since the

```
/*** kmeans ***/
for (i = start; i < stop; i++) {
    index = common_findNearestPoint(...)
    atomic [RAR] {
       *new_centers_len[index]++;
       for (j = 0; j < nfeatures; j++) {
           new_centers[index][j] +=  feature[i][j]
       }
    }
}
if (start + CHUNK < npoints) {
   atomic(RAR) { global_i += CHUNK; }
}

/*** raytrace ***/
atomic(RAR) {
   shad_ray.id = gm->rid++;
}
```

```
/*** yada ***/
while true {
   atomic(RAR) {
      elementPtr = heap_remove(workHeapPtr);
    }
    if (elementPtr == NULL) break;
    ...
}
/*** intruder ***/
while true {
   atomic(RAR) {
      bytes = stream_getPacket(streamPtr);
    }
   if (!bytes) break;
   ...
   atomic(RAR) {
     data = decoder_getComplete(decoderPtr,...);
   }
}
```

Fig. 6. Software-defined conflicts in kmeans, raytrace, yada and intruder.

root is always modified, it is impossible for several threads to concurrently remove the maximum from the same heap without suffering conflicts. Hence, this transaction would be better off if it could be executed under PCC in order to temporally serialize access to the heap. Similarly, intruder employs a queue to store the packets that conform a stream, and threads extract packets from the stream queues for processing. Popping elements from the head of a queue is also an operation known not to expose any parallelism, and thus using TM's optimistic approach to execute these critical sections concurrently becomes counter-productive. Therefore, we can use PABs to enable RAR conflicts and execute those transactions under PCC, achieving a lock-like behavior.

2.3. Avoiding False Conflicts

If the underlying TM system tracks conflicts on a cache-line granularity –as most HTM systems do–, false conflicts provoked by the conflict detection granularity can hurt application performance, leaving the programmer with few options to solve this kind of undesired behavior. The most straight-forward solution is to add padding in-between data elements so that each data falls on a different cache line. Padding is a valid strategy when there are few data elements that suffer from false sharing, but it may hurt the performance of the memory hierarchy when it affects a large portion of the data, as it happens for example in labyrinth. When a transaction adds one point to its path, concurrent attempts to update the row-adjacent points of the line will create false write-write conflicts, causing unnecessary aborts.

If the underlying TM system performs updates in place (eager data versioning, like LogTM), and supports rollbacks with word-level granularity, software-defined conflict detection can be used to mitigate false conflicts amongst transactions. Using PABs, a programmer can simply assume full responsibility on the conflict detection by disabling all conflicts types (`atomic[NONE]`), and explicitly include code that detects and resolves conflicts amongst concurrent transactions.

2.4. Avoiding Starving-Writer

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. Each client session is enclosed in a coarse-grain transaction to ensure validity of the database. The main transaction of vacation 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 and a resolution policy of requester-stalls, those transactions that attempt to update the database tables by adding or removing new items may suffer a pathological interaction commonly referred as *starving writer* [Bobba et al. 2007].

Using PABs, 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 to the reservation phase. However, a transaction never observes the database in an inconsistent state, because the PABs do not allow such transformations of conflicts, as we explain in Section 3. If in its attempt to make a reservation, a transaction re-accesses data that has changed since the query phase, it will be forced to abort.

3. LANGUAGE FEATURES FOR SOFTWARE-DEFINED CONFLICTS

In this section, we describe in detail the proposed language features to provide programmer-controlled conflicts inside transactions. As a first approach to software-defined conflicts, in this paper we investigate the association of such information with regions of code –delimited by the atomic block– rather than with data. In some cases, it may be better to identify a given memory location as being able to tolerate a given kind of interference, rather than identifying a given block of code. Nonetheless, for this study we have always kept in perspective the support of this programming construct on a hardware TM system (HTM), as our focus in order to keep the hardware changes minimal and the design as simple as possible.

3.1. PABs: Parameterized Atomic Blocks

As a programming language construct, the semantics of PABs is independent from the TM system in which the program executes. However, the underlying detection policy implemented in hardware determines the kinds of conflicts that can be observed, and thus narrows the spectrum of types that can be controlled in software. Hence, the variety of optimizations offered by PABs is restricted by when and how conflicts are detected internally, as well as by the data versioning policy used. It is important to remark that PABs are a performance-oriented construct intended for the use of experienced TM programmers who are knowledgeable about their applications as well as the TM system. If an application that uses PABs runs on top of TM system with partial or no support for software-defined conflicts, then transactions will simply execute under common transactional semantics, perhaps sacrificing some performance but always preserving correctness. In this paper, we explore the applications of software-defined conflicts in a TM system that employs eager conflict detection and eager version management, as these systems offer a wider range of conflicting interactions than their lazy counterparts, and thus more kinds of relaxations can be considered.

Definition of Conflict and Syntax. A transaction's definition of conflict is a *local* concept given by its response to each of the four possible kinds of data interactions or dependencies that a *local* transaction can experience with other *remote* concurrent transaction: read-after-write (RAW) –a remote transaction wants to read data written by the local transaction–, write-after-read (WAR) –remote write attempt to data locally read–, write-after-write (WAW) –remote write attempt to data locally written– and read-after-read (RAR) –remote read to data locally read–. The definition controls which kinds of dependencies are signalled 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 explicitly allowed using a PAB. 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 (WAW, RAW, WAR). Moreover, we use the term *empty definition* to refer to that definition in which none of the dependence types are considered to be conflicting.

The syntax of the PABs is very similar to that of regular atomic blocks. The major addition is an optional list that follows the `atomic` keyword, specifying the types of conflict to alter in the current definition.

```
atomic [ ( <dep_types> [, <xid_list> ] )  ] [<xid>] [t]
{
     BLOCK
}

<dep_types> := <dep_type> [, <dep_types>]
<xid_list> := <xid> [, <xid_list>]
<dep_type> := [!] {WAW|RAW|WAR|RAR|NONE}
<xid> := #NUMBER
```

Not every dependence type needs to be specified in each PAB, 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, indicating that such type of dependencies is removed from the definition. The block is a regular section of code grouped together, consisting in declarations and statements, for which the new definition applies. Multiple PABs can be nested. The list of transaction identifiers `xid_list` indicates which other PABs are subject to observe the software-controlled conflict definition of this PAB. `xid` is a simple label that uniquely identify this PAB in the program. The optional modifier `t` is used to force PABs to allow dependency transformations. The following paragraphs describe these modifiers in more detail.

Semantics and Scope. With the introduction of software-defined conflicts through PABs, 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 empty definition is automatically set so that conflict detection is turned off when executing non-transactional code. When the thread enters an atomic block, the default definition is implicitly set, providing the commonly-assumed concept of "two memory accesses, at least one of them a write" by default. If a list of conflict types is specified, then the current definition is updated according to the contents of the attribute given to the PAB: types of dependencies are added or removed from the current definition before executing the statements in its block, as specified. If a type appears negated in the list, it is *disabled* –removed from the definition– and not considered conflicting within the block; otherwise, that type is *enabled* –added to the definition–. Conflict types that are not specified for a given atomic block remain in the same state as outside that block (for nested atomic blocks), or as in the default definition (for the outermost PAB). Adding (removing) a type that was already enabled (disabled) has no effect.

The programmer perceives the current conflict definition as a variable with dynamic scope. All function calls inside a PAB inherit the local definition, so programmers are advised to be careful when calling code that is not aware of the relaxation. A nested PAB can redefine the definition –by making it *stricter*–, overriding the outer definition for the specified types (the rest remain unchanged). When the execution abandons a PAB, the outer definition is restored. Note that a nested PAB cannot relax

the definition established by its parent (a regular atomic block or another PAB), as that could drop isolation for its ancestors. Therefore, the use of PABs does not impact transation composability.

Compatibility and Reasoning. The introduction of PABs must maintain backwards compatibility with existing transactional code, and its introduction in programs written without having in mind software-defined conflicts must not compromise correctness. Thus, the default behavior of PABs is such that it does not force programmers to reason about the consistency property needed by all other transactions in the program: The relaxed definition is applied by default only to those data requests that belong to other dynamic instances of the same static PAB. Unless the programmer explicitly says otherwise, different PABs of the program will treat each other as regular atomic blocks, and fallback to the default definition to determine whether a potentially offending request is considered conflicting or not. Nonetheless, programmers can explicitly specify through the `xid_list` field which PABs are allowed to observe the relaxations of another PAB.

Transformation of Dependencies. By default, PABs guarantee that, once a remote transaction has trespassed a local transaction's read and write sets, the latter cannot re-access such data again throughout the course of its execution, and any attempt to do so will cause its immediate abort. For instance, when WAR dependencies are disabled, the local transaction is not allowed to reload nor modify any read set data that has been written by a remote transaction. If RAW conflicts are disabled, the local transaction is not allowed to modify any write set data that has been read by a remote transaction, including during rollback. The programmer controls whether to allow these transformations (denied by default), by means of the `t` modifier. This feature is useful when no explicit code for input consistency is included, relieving programmers from having to check if an input value about to be modified has been changed in the meantime.

4. HARDWARE AND COMPILER SUPPORT FOR SOFTWARE-DEFINED CONFLICTS

In this section, we describe the ISA and compiler extensions required to enable programmer-defined conflicts in the context of a popular hardware TM system. We take LogTM-SE as our baseline HTM system, assuming its implementation on top of a distributed directory protocol. We show that adding support for software-defined conflicts requires simple changes at both the compiler and hardware levels. Though we here explore a hardware-based solution, PABs can be also be supported and applicable in the context of software TM systems, where they can also prove beneficial.

ISA Changes. 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 [Shriraman et al. 2008; Ramadan et al. 2008], which we augment with four new bits to encode the current definition of conflict. Each type of dependency (WAR,WAW,RAW,RAR) is represented by one bit in the register, indicating whether that class is enabled –i.e. consider conflicting– in that moment. A transaction identifier field (*xid*) is also added to the status register, which contains the static *id* of the transaction, and whose content is copied to all outgoing coherence request messages. To support relaxations across different static PABs, a small table that keeps *xid_list* is required. The conflict check and book-keeping logic is slightly modified so that these four bits control how the output from the query of the read and write sets is interpreted –unlike traditional HTM systems, which embed this decision in silicon–. For any incoming request, its xid field is compared against the local xid (and each xid in the *xid_list* table, if any). If a match is found, then the four bits of the current definition control the output of the check; otherwise, the default definition

applies. The coherence protocol guarantees that a transaction observes potentially offending remote requests and it makes possible the detection of WAR, WAW and RAW dependencies without any change to the baseline system. However, a regular MESI-style directory protocol cannot directly detect RAR dependencies, as the sharers of a cache line do not observe the requests coming from new sharers. For a reader to be able to observe reads to a cache line in its read-set –i.e. detect RAR interactions–, exclusive ownership must be acquired over the line when it is first accessed. This is achieved by upgrading all requests sent to the directory from shared to exclusive, so as to invalidate all other copies in other caches. From this point, any access to the line will result in a forwarded request to the private cache whose core has enabled RAR conflicts. In other words, the bit that controls RAR conflict detection acts like an "upgrade switch" for the outgoing requests generated by transaction. In order to avoid transformation of dependencies, we incorporate 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 the PAB. When the conflict signature is not empty, it must be checked against every local load or store address until the end of the transaction, and the output from the check must be connected to the abort signal. Finally, for eager-versioned systems, the logging logic must be capable of operating with word granularity as well as lines, in order to avoid loss of committed data due to rollbacks.

Compiler Support. 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 PAB, 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 dependencies types given to the PAB. For the outermost PAB, the compiler always takes care of establishing the default definition in the status register, before setting or clearing bits according to the conflict type list. When the end of the PAB 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. The compiler also assigs a unique *xid* to each PAB if not provided.

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 [Martin et al. 2005], in conjunction with Virtutech Simics [Magnusson et al. 2002]. We use the implementation of LogTM-SE [Yen et al. 2007] 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 time-stamps. We use an ideal book-keeping scheme to track read and write sets (*perfect signatures*). Nested transations are simply flattened and subsumed to the outermost transaction, without support for partial aborts.

We perform our experiments on a tiled CMP system, as described in Table I. We use 1 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.

Table I. System Parameters

MESI Directory-based CMP	
Core Settings	
Cores	1 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
Link latency	1 cycle
Link bandwidth	40 bytes/cycle

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 six benchmarks extracted from the STAMP suite [Cao Minh et al. 2008] (genome, labyrinth, vacation, kmeans, intruder, yada), one taken from the SPLASH-2 suite [Woo et al. 1995] (raytrace) and a parallel implementation of the self organizing map algorithm [Kohonen et al. 2001] (som) that we have developed using coarse-grained transactions. All considered benchmarks were augmented using PABs according to the description of Section 2, relaxing or tightening the definition of conflict as discussed earlier. We use the small input sets for all STAMP applications except genome, for which we chose medium-size due to the very short execution time of the small set. For raytrace, the teapot input was chosen. As for the som, it works on a 25×35 matrix of four-dimensional weight vectors. Padding was added so that each 64-byte cache line contains only one weight vector. The map is trained over 500 iterations taking as input a random element from the Iris data set [Fisher 1936]. We measure parallel execution time for the fine-tuning phase of som, considering that the convergence phase begins when the neighborhood radius is 5 neurons or less.

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 through parameterized atomic blocks (PAB plot). To provide a fair comparison for benchmarks conceived to make use of early release (ER) (i.e. labyrinth, som), we augmented the baseline system with support for the removal of block addresses from the transaction read set. It is important to note that supporting ER was possible because we are simulating an ideal baseline system that uses perfect signatures –lists of addresses that precisely track the read-write sets–. ER would not be easily implemented in LogTM-SE if true hash encoding were to be used for bookkeeping. On top of that, these ER-dependant benchmarks had to be carefully developed to work properly on top an HTM that uses cache line granularity (and thus releases an entire cache line each time), so as to avoid releasing other data contained in the same entire cache line mistakenly. In that way, achieving comparable performance levels between Base and PAB when emulating ER is considered a successful result, given that the complexity of supporting PABs in LogTM-SE is much lower than supporting ER.

Furthermore, in order to compare the benefits of the PCC achieved by using PABs, we also simulate an enhanced version of LogTM-SE that employs a small write-set predictor (*Pred* plot) to selectively request exclusive permission and add the block to the transaction write-set, modelled after a migratory sharing predictor [Kaxiras and Goodman 1999]. Similar to the EE_P system described by Bobba et al. [2007], this

Table II. Number of Aborted Transactions for 16 and 32 Threads.

	16 threads				32 threads			
	Base	PAB	Pred	Diff	Base	PAB	Pred	Diff
genome	1642	625	13797	62%	3109	1456	16693	53%
intruder	38000	6974	11223	82%	48829	8916	9823	82%
kmeans-h	7578	0	67	100%	14263	0	198	100%
kmeans-l	2661	0	17	100%	15175	3	154	100%
labyrinth	758	344	735	54%	1579	705	1723	59%
raytrace	191840	28887	750	85%	290203	64288	2639	78%
som	330	252	350	24%	734	630	885	14%
vacation	112	77	162	32%	203	203	322	0%
yada	4894	3262	3840	33%	9625	6269	6841	35%

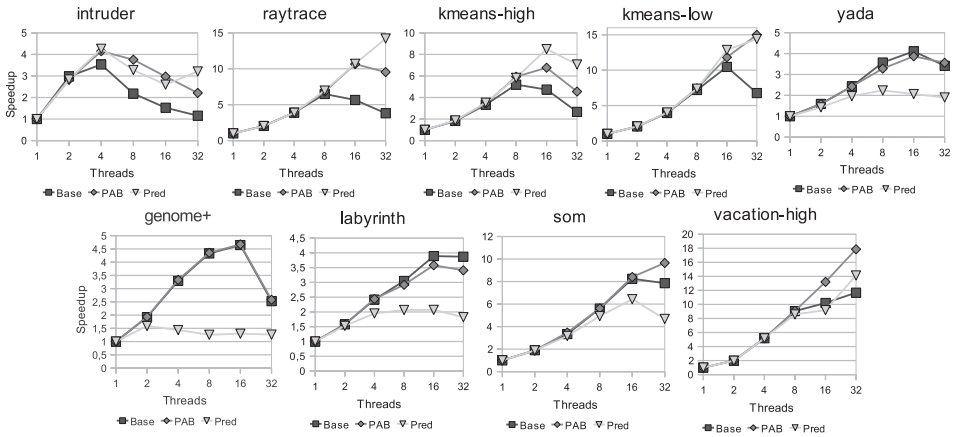

Fig. 7. Speedups over sequential code.

predictor eliminates the dueling upgrades pathology that result when transactions read, modify, and write the same block, which force one to abort in the Base system. With this optimization, the default policy of LogTM (requester-stalls) allows the transactions to serialize, mitigating the number of aborts.

5.2. Results

The main results of the evaluation are summarized in Figure 7 and Table II, which show, respectively, the speedup curves and the number of aborted transactions for the considered benchmarks and each of the three evaluated systems. For the 16- and 32-thread configurations, Table II shows the percentage of reduction in aborted transactions achieved by PABs in comparison to the Base system (columns 5 and 9), revealing that in most benchmarks software-defined conflicts are capable of eliminating more than 50% of the aborts. Figure 8 shows the normalized execution time broken down into components, for the three evaluated systems on the 16-thread configuration. In general, the results show that using PABs in the considered benchmarks can improve scalability and achieve significant performance improvements in some cases. Our results for 16 threads show that an HTM with support for PABs outperforms the baseline LogTM-SE system in nearly all benchmarks, by 18% in the average case.

Genome. Aborts suffered by concurrent insertions in the same table (dynamic instances of the same static transaction competing for the list) account for three quarters of the total number of aborts that occur in 16- and 32-thread configurations. Using

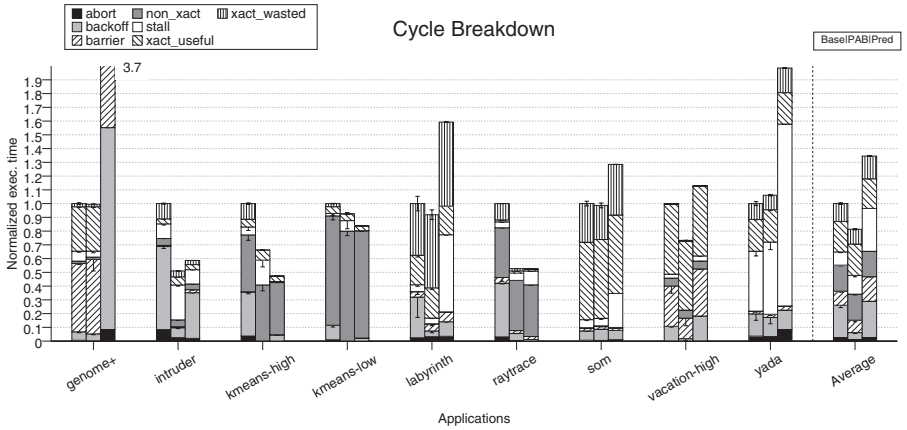

Fig. 8. Normalized execution time breakdown for 16 threads.

PABs to disable WAR conflicts for the list insert effectively eliminates more than 50% of the aborts suffered in the benchmark for these configurations, as shown in Table II. As we can see in Figure 7, the effect on the execution time is inappreciable. We find two reasons for this result: First, the phase of insertions only spans a small amount of the total execution time in genome. And second, the hardware was not augmented to ensure fairness when using PABs. In this case, it may happen that a transaction with a relaxed definition gets repeatedly aborted despite its timestamp, as any access that entails a forbidden transformation of dependencies (here, reload data remotely modified) provokes its abort. As we can observe in the Figure 8, this is reflected in the increase of the fraction of cycles spent in barrier, as well as the variability of such fraction, for the *PAB* plot when compared to *Base*. In spite of that, experiments with different random seeds revealed that when the interleaving of transactions does not create temporal starvation, the 16-thread PABs configuration reduces execution time by 10% compared to Base.

Labyrinth, som. For its main transaction, we used PABs to disable WAR conflicts in order to emulate early release, as discussed in Section 2. Using a regular PAB simplifies programming, since the path validation is not be needed: points can be directly written without further checks, as any attempt to write a point already written by an offender transaction entails a forbidden transformation of conflicts that triggers abort by default. We restored the default definition before entering this phase using a nested PAB, to ensure that once a point is observed (reloaded), its value cannnot change until the end of the transaction. Similarly, programming the som algorithm using a simple PABs to disable WAR conflicts yields a more straightforward code than the ER counterpart: With ER, threads need to validate the neighborhood before the update phase, making sure that the winner neuron and its distance to the input sample have not changed since the search phase. As we can see in Figures 7 and 8, using PABs in labyrinth and som achieves similar performance levels to the ideal Base system with support for ER, up to and including the 16-thread configuration. Furthermore, PABs allow som to scale up to 32 threads, while the Base system already starts to decline in its performance. In both benchmarks, the number of aborts is reduced when using PABs, though it is more pronounced for labyrinth (55-60%). Such reduction becomes responsible for the 8% improvement over Base for 16 threads observed in Figure 8, mostly due to the removal of the back-off portion. However, it does not have a significant effect in performance for som.

Pessimistic Concurrency Control. In regards to the applicability of PABs for PCC, both flavors of kmeans as well as raytrace prove to benefit from the execution in mutual exclusion of their transactions. Because the benchmarks are dominated by non-transactional execution, the performance gained by avoiding a lot of aborts clearly outweighs the overhead of an excessive serialization of transactions. For configurations of up to 8 threads, all three evaluated systems behave similarly because there is little contention. For raytrace and kmeans-high, Figure 7 shows how the Base system does not scale past 8 threads due to a very high number of aborts, while we observe that with PABs, these two benchmarks scale up to 16 threads and reach performance levels similar to the Pred system in the case of raytrace, before declining for 32 threads. Unlike Base and PAB, the write-set predictor of the Pred system allows raytrace to scale up to 32 threads because it solves the duelling upgrades pathology suffered by raytrace's fine-grained critical regions (read and increment a global ray id) and achieves an extraordinary reduction in the number of aborts, reflected in Table II. Using PABs also alleviates this performance pathology and reduces aborts by 78%, but the high amount of contention in the access to the ray id variable with 32 threads is not managed by PABs as efficiently as the write-set predictor does: Using PABs for PCC works like an upgrade switch so that every requested cache lines is acquired with exclusive ownership, whether or not the transaction modifies it. Thus, for each transaction in raytrace, there is an extra stall time as there are two cache lines that need to be transferred from cache to cache, the pointer to the structure that contains the global variables, `gm`, and the global ray identifier itself,*rid*. In Pred, on the contrary, only the line that contains the rid is transferred, and a shared copy of the pointer gm resides in every local cache, hence resulting in faster transactions and shorter stalls. For kmeans-low, the Base system scales up to 16 threads, and then suffers a severe performance degradation when moving to 32 threads because of the aborts. Meanwhile, using PABs to serialize the execution of the transactions makes kmeans-low scale up to 32 threads, achieving similar performance to the Pred system, whose write-set predictor successfully reduces the number of aborts by a factor of 100, as depicted in Table II. It is remarkable how using PABs to detect RAR conflicts completely eliminates aborts from both of kmeans flavors: All cache lines are requested in exclusive, including read-only data such as the pointer to the new_centers_len vector. Using RAR conflict detecting as an "upgrade switch" forces transactions in kmeans to serialize from their very first memory access. In this way, each transaction directly requests the pointer to new_centers_len vector in exclusive and does not release it until it has incremented the length in the appropriate index. The result is that transactions stall one after another without creating cycles, proving that LogTM's policy of requester-stalls is useful to resolve conflicts without the need to abort one of the transactions. The conservative approach of PABs is the reason why they are clearly outperformed by the write-set predictor in kmeans-high: despite creating few aborts, the write set predictor only upgrades to exclusive those requests that are predicted to be written (because they have been recently written), and hence transactions are never serialized in their access to read-only data, as it happens with PABs. In Figure 8 we can see that the write-set predictor manages to reduce the execution time of the 16-thread baseline system by 17% for kmeans-low, and more than 50% for kmeans-high. Although far from the performance of Pred, the PAB system also makes kmeans-high scale up to 16 threads, while the Base system begins to see a performance drop after 8 threads. As shown in Figure 8, PABs reduce execution time over 30% for kmeans-high when compared to Base for 16 threads, mainly because it eliminates the fraction of cycles spent aborting, backing-off and executing transactions that eventually abort. Obviously, now a large portion of the cycles are spent stalled for another transaction, according to the transaction chaining mentioned before.

PABs are also of use to serialize the accesses to data structures from which threads extract jobs for processing, as it is the case of yada and intruder. Although the execution of intruder's "queue pop" transactions in mutual exclusion significantly reduces the number of aborts, the speedup graphs show that there is little difference in terms of execution time with the Base system. This is because these aborts of are not the real bottleneck that stops the application from achieving better performance. However, the inability of the Base system to deal with high contention to these queues contributes even more to the performance degradation as we scale up the number of threads. Running intruder with PABs is 15% faster than the baseline for the best configuration (4 threads), and then the performance of both systems slowly degrades as we move on to larger number of threads. In this benchmark, PABs perform almost two times faster than the baseline for 16 threads (see Figure 8), because of the 82% reduction in the number of aborts, responsible for the huge portion of back-off cycles that we observed in the baseline case. The abort and wasted components are also reduced, and in turn the stall fraction increases due to the chaining of transactions. All aborts corresponding to the queue pop transactions (stream_getPacket and decoder_getComplete) are completely removed by PABs, leaving only the aborts caused by the main transaction of intruder (decoder_process), whose conflicts entail the actual performance limits of this benchmark. In yada, the use of PABs does not improve execution time compared to the Base case because the pointer to the work heap is heavily used inside other transactions aside from the heap_remove transaction that we wrap inside a PAB. The result is that besides serializing the extraction of work elements from the heap, we are creating more conflicts with other transactions (region_transferBad) that also access to the work heap, increasing the number of aborts suffered by the latter. Thus, yada is an example of scenario in which PABs must be used carefully, since the variables accessed inside the PAB are also accessed from other regular atomic blocks that we are not interested in executing under PCC. Finally, when evaluating all the benchmarks with the enhanced LogTM system that uses a write-set predictor shows that this kind of hardware device is useful to detect migratory patterns and avoid dueling upgrades in some scenarios –i.e. raytrace, kmeans–, but its use does not always benefit performance. In this way, using PABs as a programmer-accessible upgrade switch seems to be a more flexible and efficient approach to solve the pathological interactions suffered by highly contended transactions, rather than relying on a transparent but fixed hardware component like the write-set predictor, whose gains are not generalizable, as we see in Figure 7 for benchmarks like yada, genome or labyrinth.

Vacation. We observe in Figure 7 how performance scales close to the ideal up until 8 threads for both Base and PAB configurations, because of a very low or nonexistent number of aborts. For the 16-thread experiment, the number of aborts goes up by a factor of 15-20 when compared to 8 threads, and this is reflected on a performance degradation. The aborts occur because a writer transaction suffers starvation when it tries to add or remove items from the database in the present of readers, as readers impede the writer to gain exclusive access to the block, due to LogTM's resolution policy of nack. Figure 8 shows how the fraction of back-off cycles is considerable for the Base plot, while nonexistent for PAB; the stall fraction also disappears with PAB, and the barrier portion gets considerably decreased. These numbers quantitatively reflect that PABs solve the aforementioned pathology.

6. RELATED WORK AND DISCUSSION

Several programming language constructs motivated by performance optimization have been proposed for TM. Early release (ER) [Skare and Kozyrakis 2006] 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, programmers must be careful about 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 a very low-level construct whose use entails a difficulty that resembles a lot that of fine-grained locks. Second, it is up to the programmer to avoid or deal with inconsistent input data (a different data version observed if it reloads data previously released) Furthermore, ER only allows to remove elements from the read set, but not from the write set. On the hardware side, ER is not easy to support in all HTM systems. Systems that use hash signatures do not have the possibility of "removing" an individual address. Using RW-bits in the private cache makes ER 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. Furthermore, ER is difficult to implement consistently in HTM systems that use cache line granularity, since an ER instruction expects a word address and thus it is not safe to release the entire cache line.

Open nesting [Ni et al. 2007] 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. Thus, 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 flattening 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, 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, 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 [Herlihy and Koskinen 2008], 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 [Shriraman et al. 2008] 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).

The goal of this work is to study the benefits of introducing flexibility along the conflict detection dimension in a pure HTM implementation, rather than coming up with a novel TM programming construct that opens yet another way of relaxing the memory consistency model. We adopt a bottom-up approach when we decide to replace the typical immutable definition of conflict embedded in previously proposed HTMs, with a configurable one. When exploring the different opportunities offered by such

hardware design choice, we address the challenge of how to reduce conflicts between transactions amongst other applications, but we do so from the perspective of a HTM solution that assumes no software intervention whatsoever. In fact, the value of the proposed language construct is precisely its straightforward implementation in silicon: We leverage an existing HTM implementation and then modify it slightly to offer a simple and clean interface to the upper layers for controlling the newly configurable hardware component (i.e. the conflict detection module). We propose PABs as the interface that allows the programmer to directly communicate with the conflict detection logic, and then show how PABs can be used to control the definition of conflict, so as to reduce the amount of aborts suffered by a transactional application.

Prior work found in the STM literature, such as transactional collection classes [Carlstrom et al. 2007], transactional boosting [Herlihy and Koskinen 2008] or open nesting [Ni et al. 2007], raise the level of abstraction at which a TM system operates, introducing higher-level notions of conflict detection. Unlike them, PABs are not intended for high-level conflict detection and their suitability for implementing concurrency control at the semantic level is arguable. Given the fact that PABs are directly supported by hardware, the low level of abstraction is inherent to their nature. Hence, a programmer using PABs can only work with memory-level (physical) conflicts and their type –depending on the transaction interleaving–, but still ignores abstract (logical) conflicts and remains oblivious to concepts such as commutativity. While defining conflicts at the level of data structure operations or other high level constructs that make sense to the programmer is a much easier programming approach, these proposals typically require significant software run-time support to implement the functionality (i.e. abstract locks for semantic dependency tracking, commit and abort handlers, etc.), and thus the applicability of this ideas is restricted to the STM domain. In spite of their fundamental limitation regarding the abstraction level, PABs demonstrate that having a software-controlled definition of conflict can be benefitial in the context of high performance HTM systems, so that expert programmers can take advantage of them for tunning-up their codes in search for more performance. Since PABs do not rely on software runtimes that enrich their semantics, comparisons with prior work on semantic concurrency control in terms of complexity of reasoning, programmability and ease of debugging may lead to biased conclusions. Compared against STM-based techniques, PABs do appear as a more difficult and error-prone programming construct than other schemes like transactional collection classes, open nesting or transactional boosting.

7. 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 the paramterized atomic block (PAB) as a new programming construct that enables the programmer to alter the definition throughout the execution of a transaction. PABs are used to relax the conditions for signaling a conflict in those cases where relaxing the isolation guarantees does not threaten correctness, in order to eliminate conflicts that are not essential. We have shown how PABs can also be used by programmers to enforce conservative synchronization in cases where optimistic concurrency control proves counterproductive.

We have presented several examples of its use on common scenarios and well-known benchmarks. We have described an implementation of this construct on top of an eager HTM system. Finally, we have evaluated the proposed hardware/software scheme on top of a popular HTM system, 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

Ananian, C. S., Asanovic, K., Kuszmaul, B. C., Leiserson, C. E., and Lie, S. 2005. Unbounded transactional memory. In *Proceedings of the 11th Symposium on High-Performance Computer Architecture*. 316–327.

Bobba, J., Moore, K. E., Yen, L., Volos, H., Hill, M. D., Swift, M. M., and Wood, D. A. 2007. Performance pathologies in hardware transactional memory. In *Proceedings of the 34th International Symposium on Computer Architecture*. 81–91.

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

Carlstrom, B. D., McDonald, A., Carbin, M., Kozyrakis, C., and Olukotun, K. 2007. Transactional collection classes. In *Proceedings of the ACM SIGPLAN Conference on Principles and Practice of Parallel Computing*. 56–67.

Fisher, R. A. 1936. The use of multiple measurements in taxonomic problems. *Annals Eugen. 7*, 179–188.

Hammond, L., Wong, V., Chen, M., Carlstrom, B. D., Davis, J. D., Hertzberg, B., Prabhu, M. K., Wijaya, H., Kozyrakis, C., and Olukotun, K. 2004. Transactional memory coherence and consistency. In *Proceedings of the 31st International Symposium on Computer Architecture*. 102–113.

Herlihy, M. and Koskinen, E. 2008. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 207–216.

Hill, M. D., Hower, D., Moore, K. E., Swift, M. M., Volos, H., and Wood, D. A. 2007. A case for deconstructing hardware transactional memory systems. Tech. rep., Univ. of Wisconsin CS-TR-2007-1594.

Kaxiras, S. and Goodman, J. R. 1999. Improving cc-numa performance using instruction-based prediction. In *Proceedings of the 5th Symposium on High-Performance Computer Architecture*. 161.

Kohonen, T., Schroeder, M. R., and Huang, T. S., Eds. 2001. *Self-Organizing Maps*.

Magnusson, P. S., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A., and Werner, B. 2002. Simics: A full system simulation platform. *IEEE Computer 35,* 2, 50–58.

Martin, M. M., Sorin, D. J., Beckmann, B. M., Marty, M. R., Xu, M., Alameldeen, A. R., Moore, K. E., Hill, M. D., and Wood, D. A. 2005. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *Comput. Architect. News*, 92–99.

McDonald, A., Chung, J., Brian, D. C., Cao Minh, C., Chafi, H., Kozyrakis, C., and Olukotun, K. 2006. Architectural semantics for practical transactional memory. In *Proceedings of the 33rd International Symposium on Computer Architecture*. 53–65.

Moore, K. E., Bobba, J., Moravan, M. J., Hill, M. D., and Wood, D. A. 2006. LogTM: Log-based transactional memory. In *Proceedings of the 12th Symposium on High-Performance Computer Architecture*. 254–265.

Moravan, M. J., Bobba, J., Moore, K. E., Yen, L., Hill, M. D., Liblit, B., Swift, M. M., and Wood, D. A. 2006. Supporting nested transactional memory in LogTM. In *Proceedings of the 12th International Symposium on Architectural Support for Programming Language and Operating Systems*. 359–370.

Ni, Y., Menon, V. S., Adl-Tabatabai, A.-R., Hosking, A. L., Hudson, R. L., Moss, J. E. B., Saha, B., and Shpeisman, T. 2007. Open nesting in software transactional memory. In *Proceedings 12th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*. 68–78.

Ramadan, H. E., Rossbach, C. J., Hofmann, O. S., and Witchel, E. 2008. Dependence-aware transactional memory. In *Proceedings of the 41st International Symposium on Microarchitecture*. 246–257.

Shriraman, A., Dwarkadas, S., and Scott, M. L. 2008. Flexible decoupled transactional memory support. In *Proceedings of the 35th International Symposium on Computer Architecture*. 139–150.

Skare, T. and Kozyrakis, C. 2006. Early release: Friend or foe? In *Proceedings of the Workshop on TM Workloads*.

Sonmez, N., Harris, T., Cristal, A., Unsal, O. S., and Valero, M. 2009. Taking the heat off transactions: Dynamic selection of pessimistic concurrency control. In *Proceedings of the 23rd International Parallel and Distributed Processing Symposium*. 1–10.

Volos, H., Goyal, N., and Swift, M. 2008. Pathological interaction of locks with transactional memory. In *TRANSACT '08: 3rd Workshop on Transactional Computing*.

WATSON, I., KIRKHAM, C., AND LUJAN, M. 2007. A study of a transactional parallel routing algorithm. In *Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques*. 388–398.

WOO, S. C., OHARA, M., TORRIE, E., SINGH, J. P., AND GUPTA, A. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*. 24–36.

YEN, L., BOBBA, J., MARTY, M. R., MOORE, K. E., VOLOS, H., HILL, M. D., SWIFT, M. M., AND WOOD, D. A. 2007. LogTM-SE: Decoupling hardware transactional memory from caches. In *Proceedings of the 13th Symposium on High-Performance Computer Architecture*. 261–272.

Received July 2011; revised October 2011; accepted November 2011