LV*: a low complexity Lazy Versioning HTM infrastructure

Anurag Negi, MM Waliullah, Per Stenstrom
Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden

Abstract— Transactional memory (TM) promises to unlock parallelism in software in a safer and easier way than lock-based approaches but the path to deployment is unclear for several reasons. First of all, since TM has not been deployed in any machine yet, experience of using it is limited. While software transactional memory implementations exist, they are too slow to provide useful experience. Existing hardware transactional memory implementations, on the other hand, can provide the efficiency required but they require a significant effort to integrate in cache coherence infrastructures or freeze critical policy parameters.

This paper proposes the LV* (lazy versioning and eager/lazy conflict resolution) class of hardware transactional memory protocols. This class of protocols has been implemented with ease of deployment in mind. LV* can be integrated with low additional complexity in standard snoopy-cache MESI-protocols and can be accommodated in a directory-based cache coherence infrastructure. Since the optimal conflict resolution policy (lazy or eager) depends on transactional characteristics of workloads, LV* supports a set of conflict resolution policies that range from LazEr – a family of Lazy versioning Eager conflict resolution protocols – to LL-MESI which provides lazy resolution. We show that LV* can be hosted in a MESI protocol through straightforward extensions and that the flexibility in the choice of conflict resolution strategy has a significant impact on performance.

Keywords: Hardware Transactional Memory, Parallel Architectures

I. INTRODUCTION

Transactional memory (TM) [11] as a programming paradigm promises safer and easier extraction of concurrency from multithreaded applications when compared to fine-grained lock-based approaches. There is a broad consensus that constructs that use transactional abstractions are more natural for programmers to reason about and can potentially provide reasonable performance. Yet, experience is limited since TM has not been deployed broadly.

Software transactional memory is a relatively mature field and some embodiments have been deployed in production-quality compilers [18]. While they provide a framework for gaining experience working with transactions, the loss in performance is too big to attract a reasonable user community [3]. Hybrid TM systems provide relatively inexpensive hardware support for speeding up some routine TM actions like data race detection but implement critical concurrency managers in software [25]. A related category of TM implementations exists [6], AMD ASF [7] where extremely limited hardware support exists for running tiny critical sections concurrently. Such schemes, through careful use, can be used to speed up STM implementations but the resources available are too limited to support common use cases purely in hardware.

Hardware transactional memory (HTM) proposals aim to bring the speed of silicon to TM. Two seminal works, namely LogTM [17] and TCC [8], have explored two ends of the HTM design space, which can be broadly described by two characteristics, namely version management and conflict resolution. Version management deals with the manner in which visibility of speculative updates made by transactions is controlled. While protocols employing lazy version management and conflict resolution, such as TCC, do not propagate changes to shared memory until commit time, protocols employing eager version management and resolution, like LogTM, perform immediate inplace updates. They expose different tradeoffs both concerning performance and complexity. In terms of performance, while lazy versioning together with conflict resolution puts the burden on commits to update shared memory, which may impose a scalability limit, eager version management results in performance loss at aborts to restore the shared memory state. From a complexity standpoint, TCC is not compatible with standard snoopy-based cache coherence infrastructures (e.g. MESI) and LogTM must support undo logs to restore system states which adds to the complexity.

Conflict resolution is another dimension of the HTM design space in which TCC and LogTM differ. While TCC resorts to lazy conflict resolution, LogTM resolves conflicts eagerly. While many studies [2, 24] have considered the tradeoff between lazy and eager conflict resolution, results are mixed and it is still an open question what policy to use. Hence, freezing such policy decisions in the protocol specification should be avoided. In conclusion, what is needed is an HTM implementation that can be integrated in 1) a standard cache protocol infrastructure with low complexity and performance loss, 2) that supports a multitude of conflict resolution policies, and 3) that can scale up with core count.

This paper presents the LV* class of HTM protocols. It uses lazy versioning (LV) to avoid costly aborts. By detecting conflicts
eagerly we present a protocol innovation that in a distributed
fashion keeps track of what transactions have had a conflict with
a committing one which implements efficient commits. Through
its eager conflict detection it can leverage on the data race
detection supported by the underlying MESI cache coherence
protocol which drastically reduce its complexity. Moreover, it
keeps the policy by which conflicts are resolved open for
software to decide (in LV* denotes this freedom). We show
that a number of policies can be supported with respect to conflict
arbitration (selecting a winning transaction on a conflict) and
conflict rectification (selecting a policy for resolving the
conflict). LV* supports a family of eager conflict resolution
policies called LazEr – Lazy version management/Eager conflict
resolution – and a lazy conflict resolution protocol called LL-
MESI. We show that all these protocol instances can be
supported with minor extensions to a snooping-based MESI protocol
and provides a scalable path to directory protocols.

We make the following contributions:

- We propose LV*, a novel implementation of lazy versioning
  HTMs that can be integrated with low complexity in a
  standard cache coherence protocols.
- We show how a set of conflict resolution policies ranging
  from eager to lazy resolution can be hosted in a single
design.
- LV* includes a novel lazy conflict resolution configuration
designed to greatly simplify global commit operations.
- We evaluate a set of conflict resolution policies and
  highlight the importance in terms of not freezing policy
decisions in HTM protocols.

Section II describes the LV* protocol and issues related to its
integration in the baseline multicore framework. Section III
details our experimental methodology. We provide results and
analysis of the design in Section IV. Section V describes related
work. Section VI concludes the paper.

II. ARCHITECTURAL FRAMEWORK

Section II(A) describes the baseline multicore architecture used
as a substrate for implementing transactional extensions. Section
II(B) through Section II(C) describe the LV* protocol and its
implementation on a single cluster. Section II(D) discusses how
the TM system can be configured by an application at runtime.
Section II(E) deals with issues concerning scalability.

A. The Baseline Multicore System

A cluster-based architecture is chosen for the baseline system
as depicted in Fig. 1 to facilitate a smooth path toward many-core
scalability. Each cluster includes processing cores, per core
private non-blocking L1 caches, an inclusive shared L2 cache and a
split-transaction bus with independent snoop and data
transfer that uses a MESI cache protocol to maintain coherence.
An inclusive L2 cache is chosen to allow easy interconnection of
clusters communicating via shared memory.

B. The LV* TM Protocol

1) LazEr: Flexible Eager Conflict Resolution:

LazEr is a flexible Lazy versioning and Eager conflict
resolution HTM design. We chose lazy as opposed to eager
versioning as it simplifies actions needed to clean up the system
state on an abort. Since lazy versioning holds speculative data
within the private cache, it can be quickly discarded when
required (e.g. via gang clearing cache metadata or resetting
buffers). The choice of eager conflict resolution is motivated by
the fact that it eliminates the need for global arbitration at commit
time. At the point of transaction commit the thread already owns
all the lines it has modified.

A novel aspect of LazEr is that it does not freeze the conflict
resolution policy. It considers conflict resolution as comprising
two distinct actions, namely conflict arbitration and conflict
rectification. Conflict arbitration involves choosing a winner
when a conflict is detected. Conflict rectification is the corrective
action that is required after conflict arbitration has yielded a
result. Conflict arbitration uses various transaction priorities
(described later) to choose the winner. LazEr allows the priority
scheme to be chosen at runtime to suit application characteristics.

Conflict rectification can either stall a transaction or abort it.
Stalling a transaction can save wasted work but it introduces a
requirement that the stalled transaction be notified when the
winner aborts or commits. However, stalling a transaction is

![Figure 1. The baseline cluster-based multiprocessor organization used for incorporation of LazEr and LL-MESI.](image-url)
possible only when the requester (remote transaction) loses the conflict. As we shall show later, stalling does not always perform better than simply aborting all transactions that lose conflicts. Hence, LazEr also allows conflict rectification policy to be chosen at runtime.

2) LL-MESI: Fast Lazy Conflict Resolution:

Studies exist [24] that show lazy conflict resolution is quite efficient at extracting available concurrency in several transactional workloads. Lazy conflict resolution can be incorporated into the framework by permitting multiple caches to buffer speculatively modified copies of the same cache line. When such copies exist, the protocol provides the last committed versions of cache lines to requests from private caches. Since conflicts are detected eagerly through the MESI protocol, conflicts can be recorded as they happen. A novel insight in LL-MESI is that the retention of information concerning conflicts seen over the lifetime of a transaction allows global commit action (e.g. a write-set broadcast or gaining exclusive ownership of speculatively modified lines) necessitated by lazy conflict resolution to be performed very quickly as no write-set broadcast is required.

The next section shows how the novel aspects of LazEr and LL-MESI can be integrated at low cost in a MESI cache coherence infrastructure.

C. LV* Implementation

1) Integration of LazEr in a MESI infrastructure:

LazEr leverages MESI cache-coherence for conflict detection. Fig. 2 depicts MESI state transitions and explicitly shows conditions under which conflict arbitration and rectification occur. These protocol actions incorporate the flexibility provided by LV* and can be designed without much impact on the complexity of the coherence protocol. M, E and S states are annotated with speculatively read (SR) and speculatively written (SW) status indicators. Only M lines can have the SW indicator set. E and S lines can have only the SR indicator set. Speculative updates to dirty (if M or M with SR is set) lines inside transactions cause old data to be written back to the shared L2 cache. These annotations give rise to four new protocol states, namely Mr, Msr, Esr and Ssr, as can be seen in Fig. 2, when running transactions and enable conflict detection to occur when incoming snoop messages hit such lines in private L1 caches. A conflict occurs in the following cases:

a) an incoming write or exclusive read snoop hits a line with either SR or SW indicator set
b) an incoming read snoop hits an M line with SW indicator set

When a transaction commits it clears all speculative indicators in the cache whereas, on an abort, SR indicators are cleared and all M lines with SW indicator set are invalidated. No global actions are required in either case. These cache line annotations are not unique to our proposal and have been employed by other schemes [1,4,8,17] to achieve similar aims. Any non-transactional snoop that hits a speculatively accessed line causes a transaction abort.

Figure 2. Details of cache coherence implementation for LazEr. Arrows directing towards the periphery indicate bus accesses while those pointing inwards from it indicate responses to previous bus accesses initiated by the L1 cache.
Priorities are assigned to transactions in a manner which avoids deadlock or livelock conditions. The priority assignment policy can be configured at runtime to use different schemes ranging from static timestamp priorities to dynamic ones based on work done. The core keeps track of the priority in a special register which we refer to as htm_priority_reg.

Static priorities are retained throughout the lifetime of a transaction. One example of static priorities is the use of timestamps [19,20]. A unique timestamp is assigned to each transaction locally at the time of its incarnation. Older transactions have a higher priority. While decentralized timestamp generation is important for a scalable system it is not trivial to generate a system wide unique timestamp locally. LazEr, when using static priorities, uses htm_priority_reg to maintain the number of total transactions started in the system so far. It is incremented by one whenever a new transaction starts locally and updated as soon as possible after a higher value has been seen in an incoming snoop request. Since information regarding a new transaction is known to other processors only after snoop exchange occasionally two conflicting transactions might be assigned the same value. A unique core identification number used as the least significant field of the priority resolves the race.

Dynamic priorities change as a transaction proceeds. The idea here is to use some measure of the progress made by a transaction as its priority [24,26]. Two ideas have motivated us to employ this policy – 1) the fact that a transaction that has executed fewer instructions will waste less work when aborted, and 2) the intuition that a transaction that has done more work is more likely to commit successfully and sooner. In workloads where similar sized transactions compete for shared resources, this measure can also indicate the amount of work left before commit. LazEr uses the number of instructions executed from the beginning of the transaction as priority. This scheme turns out to be significantly different than the use of fixed timestamps, especially for large long running transactions that frequently stall for memory accesses or on exceptions. In such situations using a dynamic priority helps in choosing a better alternative when a conflict occurs. A register stores the number of executed instructions since the start or restart of a transaction. The register is reset when a transaction begins or aborts and incremented whenever an instruction completes. LazEr keeps track of these metrics for each transactional thread in htm_priority_reg.

Though fixed priorities can be communicated at the start of a transaction using a special message, a similar mechanism seems infeasible with dynamic priorities. Hence, we choose to communicate transaction priorities to other threads using dedicated sideband signals on the snoop bus.

We employ a special sideband conflict signal on the snoop bus to notify a remote transaction when it loses a conflict. Local conflict rectification must cause an abort, but it waits for the snoop response to determine whether such an action is really needed. It may not be required if the remote transaction is asked to perform rectification by a transaction on another core. Conflict rectification at remote transaction may result in stalls based on the chosen configuration. Transactions that have stalled others broadcast a special notification on the snoop bus when they commit or abort. This allows the suspended transaction to resume execution.

The list below summarizes the changes required for incorporating LazEr in the baseline architecture:

1) Incorporating LL-MESI:
   - Protocol support for commit and abort notices on the snoop bus
   - Cache line state annotations using SR and SW indicators

2) Incorporating LL-MESI:
   - The LL-MESI implementation builds on LazEr to allow lazy conflict resolution to take place. The coherence infrastructure provides information pertaining to conflicts between transactions. Mechanisms are put in place in the private caches to track this information over the lifetime of a transaction. The caches now maintain, besides LazEr additions, a killer map (as has been used by EazyTM [27]) and a new cache-line annotation bit, RCONF. The killer map is a bit-map indicating cores with which a transaction has seen conflicts. The RCONF indicator is set for addresses that have been speculatively read by a transaction but have been speculatively modified by others. This allows us to selectively invalidate only those speculatively read lines that might have been modified by the committing transaction. A novel insight is that the committing transaction does not need to broadcast its write set.

When a transactional snoop request hits a line in the M state with the SW indicator set, a write-conflict is indicated to the snoop initiator. LazEr’s priority computation lines on the snoop bus are re-employed by LL-MESI to provide per-core write conflict indication. The snoop initiator then records the conflict by setting the corresponding bit in its killer map. If the access was a read, the RCONF indicator for the cache line is set. Similar actions happen when write snoops hit lines with SR bits set. When such conflicts are indicated, the shared L2 cache provides the last committed data for the cache line. This is a departure from traditional cache-coherence used by LazEr.

When a transaction commits, it simply broadcasts a commit message. All transactions that had added this transaction to their killer maps immediately abort. They reset the killer map and invalidate all lines that have the RCONF indicator set, in addition to those in M state with the SW bit set. If the aborting transaction had previously indicated a write-conflict to another transaction it broadcasts a special abort message on the snoop bus causing the corresponding bit in the killer-maps of other concurrently running transactions to be reset. The committing transaction simply clears any speculative annotations and resets both the killer map and its RCONF indicators. Both commit and abort messages utilize the protocol support that was added to allow
transaction stalls in LazEr. To reiterate, as a transaction executes concurrently, its killer map indicates cores that have speculatively updated some lines that it has read or written to, while the RCONF indicator is set for addresses that need to be invalidated, in addition to speculative updates (which are annotated using the SW indicator), when a miss-speculation occurs. The example in Fig. 3 illustrates how the LL-MESI mechanism operates.

A subtle issue that should be mentioned here is the case when a line with RCONF set is evicted and read back again in the same transaction. On eviction its address gets added to a bloom filter implementation referred to as the read set tracker (discussed in Section II(E.1)). All L1 allocations in transactions are checked for hits in the read-set-tracker. If a hit is detected the allocated line’s RCONF indicator is set pessimistically. In summary, LL-MESI requires two new structures that were not part of the LazEr framework, namely the killer-map and the RCONF indicator. It also requires the existence of a read set tracker to handle rare cases of lines that are read back after eviction.

D. Runtime HTM Configuration

The HTM extensions can be configured on a per-application basis. Each core has programmable registers to inform the hardware of the conflict resolution policy chosen. LazEr can be safely reconfigured at any point when no thread of an application is executing transactions.

All cores being used by an application must be configured with the same parameters. The configuration is part of the application’s state. Different applications can use different configurations as long as they do not share data transactionally. Provided below is a list of options that we have studied in this work:

<table>
<thead>
<tr>
<th>Conflict Arbitration</th>
<th>LazEr-Timestamp/LazEr-Dynamic(Work Done)/LL-MESI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conflict Rectification</td>
<td>Abort/Stall (LL-MESI Arbitration forces this field to ABORT)</td>
</tr>
</tbody>
</table>

E. Scalability

1) Transaction Size:

Lazy versioning using private caches sets an upper bound of the quantity of data can be accessed by one transaction. While private caches can buffer substantial amounts of such data, a safety net must be provided that guarantees correctness and forward progress when they prove insufficient. This can be implemented in software or by the hardware, e.g. through the acquisition of a global lock by a recalcitrant transaction. While such a measure of last resort must be available, some relatively simple additions can make such events as infrequent as possible. We attempt to tackle the problem of limited read-set capacity by implementing a read-set tracker. It is a bloom filter, as has been used by other HTM proposals for storing read/write set signatures [4,29]. Unlike previous proposals, in LV* addresses of speculatively read lines are added to the read-set tracker only when they are evicted from private caches. This simple addition allows much larger transactions to run without resorting to

![Figure 3. At time T0 the killer-maps (KMap) on all cores are as shown. By time T2 cores 2 and 4 have read cache line A. At time T3 core 1 writes to cache line A prompting cores 2 and 4 to set core 1 as killer and sets RCONF indicator for A. At T5 core 3 tries to read cache line B which had been speculatively modified by core 2 at time T4. The write conflict indication causes core 3 to mark core 2 as killer and sets RCONF indicator for B. At T6 core 1 commits which results in aborts on cores 2 and 4 causing invalidation of lines with RCONF set. The abort notification sent by core 2 at T7 clears the corresponding bit in core 3’s killer map.](image-url)
The addition of a read set tracker also provides a neat way of preserving information on lines with the RCONF indicator set that need to be evicted (discussed in Section II(C)). Limited write-set evictions due to limited associativity in private caches are buffered in a special victim cache. The contents are written back on commit or discarded on miss-speculation. To allow for transactions of arbitrary size and duration an orthogonal transaction virtualization mechanism could be employed, as has been explored by Rajwar et al. in VTM [20]. Further exploration of such mechanisms is not part of the scope of this work.

2) Number of cores:
Sections II(C) described how LazEr and LL-MESI can be incorporated into a single cluster containing cores, their private L1 caches and the shared L2 cache. We now briefly describe a way in which several such clusters could cooperate in a shared memory multiprocessing environment. To enable applications to run transactions in different clusters a distributed directory based cache coherence protocol is used. Each cluster is augmented with a directory bank corresponding to a region of the shared physical address space. Private caches within a cluster are made aware when lines are shared across clusters. This is done by annotating such lines with a multi-cluster-share indicator. Cores wishing to modify such lines can only do so after the L2 cache controller indicates that other clusters have invalidated their copies. While running transactions, priorities are piggy-backed on directory protocol messages. The directory protocol, just like the intra-cluster snoop protocol, is modified to convey conflict resolution indications. L2 cache controllers now implement logic to detect conflicts between buffered accesses and cause local miss-speculation if required.

### TABLE 1. SIMULATOR CONFIGURATION

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Configuration/Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processors</td>
<td>8 cores each running at 2GHz</td>
</tr>
<tr>
<td>L1 Parameters</td>
<td>64KB, 8 way, 32 byte line size, LRU replacement, latency 2 cycles</td>
</tr>
<tr>
<td>L2 Parameters</td>
<td>2MB, 4 way, 32 bytes line size, Random replacement, latency 20 cycles</td>
</tr>
<tr>
<td>Memory access latency</td>
<td>200 cycles</td>
</tr>
<tr>
<td>Operating System</td>
<td>Solaris 10</td>
</tr>
<tr>
<td>Architecture</td>
<td>Sparc V9</td>
</tr>
</tbody>
</table>

### III. EXPERIMENTAL METHODOLOGY

Our experimental setup comprises a memory system simulator hooked to Simics. Simics [13] is a full system simulator with support for multiprocessor simulation. Our configuration runs Solaris 10 on a SPARC V9 SMP cluster. The memory system simulator implements cycle accurate cache models and detailed intra-cluster communication. The L2 controller models latencies for L2 hits and misses serviced by physical memory. The analysis presented in the sections that follow is based on simulations performed on 8 cores in a single cluster configuration. Table 1 contains details of various simulator parameters.

We have used STAMP workloads [15] for evaluating various architectural configurations. They comprise 8 different transactional benchmarks. Some like kmeans and SSCA2 have very short transactions and are highly data parallel. Bayes and yada have larger transactions with a lot of contention. Bayes, too, has large transactions but its utility as a benchmark is questionable as runtimes vary significantly with minor changes in initial conditions. Genome, intruder and vacation have moderate sized transactions but do not exhibit a lot of conflicts. Simics’ magic instructions have been used to annotate transaction boundaries in these applications.

### TABLE 2. STAMP APPLICATION PARAMETERS

<table>
<thead>
<tr>
<th>Application</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kmeans</td>
<td>-m40 -n40 -t0.05 -i random-n2048-d16-c16.txt</td>
</tr>
<tr>
<td>SSCA2</td>
<td>-s13 -l1.0 -u1.0 -l3 -p3</td>
</tr>
<tr>
<td>Intruder</td>
<td>-a10 -l4 -n2048 -s1</td>
</tr>
<tr>
<td>Genome</td>
<td>-g256 -s16 -n16384</td>
</tr>
<tr>
<td>Vacation</td>
<td>-n2 -q90 -u98 -r8192 -t4096</td>
</tr>
<tr>
<td>Yada</td>
<td>-a20 -i633.2</td>
</tr>
<tr>
<td>Labyrinth</td>
<td>-irandom-x32-y32-z3-n96.txt</td>
</tr>
<tr>
<td>Bayes</td>
<td>-v16 -r1024 -n2 -p20 -i2 -e2</td>
</tr>
</tbody>
</table>

### IV. EXPERIMENTAL RESULTS

We now present results of our experiments with the architectural framework described in Section II. Section IV(A) deals with transactional characteristics for STAMP workloads running on the available HTM configurations. We feel this is important to view later results in the correct perspective. Sections IV(B) and IV(C) analyze the impact of the choice of conflict arbitration and conflict rectification policies respectively. Section IV(D) then compares the sensitivity of HTM performance to variations in architectural parameters.
A. Benchmark Characteristics

Table 2 shows some pertinent characteristics of STAMP benchmarks comparing metrics for LazEr to those for LL-MESI. In this case LazEr employs simple timestamp based priority with abort-on-conflict rectification. Transactional residency tells us the percentage of the total parallel execution time an application spends executing transactions. The read/write set sizes represent the 90th percentile size (in cache-lines) among committed transactions. Wasted work metrics represent the average number of instructions that are rolled back after an abort. These metrics are provided to assist the reader in appreciating the differences in transactional behavior of workloads.

![Variation in transactional behavior of STAMP applications](image)

Figure 4. Characterization of STAMP benchmarks on the basis of contention and transactional residency

Table 2. Some Pertinent Numbers for STAMP Benchmarks

<table>
<thead>
<tr>
<th></th>
<th>Transactional Residency (%)</th>
<th>Read Set Size</th>
<th>Write Set Size</th>
<th>Aborts per Commit (LazEr: TS/Abort)</th>
<th>Aborts per Commit (LL)</th>
<th>Wasted work per Abort (Eager CR-LE Baseline)</th>
<th>Wasted work per Abort (LL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yada</td>
<td>97</td>
<td>248</td>
<td>127</td>
<td>103</td>
<td>0.32</td>
<td>73</td>
<td>25628</td>
</tr>
<tr>
<td>Labyrinth</td>
<td>90</td>
<td>771</td>
<td>445</td>
<td>570</td>
<td>4.02</td>
<td>3034</td>
<td>614386</td>
</tr>
<tr>
<td>Vacation</td>
<td>76</td>
<td>73</td>
<td>9</td>
<td>0.58</td>
<td>0.57</td>
<td>279</td>
<td>748</td>
</tr>
<tr>
<td>Genome</td>
<td>60</td>
<td>86</td>
<td>4</td>
<td>0.24</td>
<td>0.06</td>
<td>701</td>
<td>1805</td>
</tr>
<tr>
<td>Bayes</td>
<td>60</td>
<td>229</td>
<td>113</td>
<td>83</td>
<td>0.75</td>
<td>722</td>
<td>18112</td>
</tr>
<tr>
<td>Intruder</td>
<td>34</td>
<td>28</td>
<td>9</td>
<td>2.49</td>
<td>0.27</td>
<td>33</td>
<td>348</td>
</tr>
<tr>
<td>Kmeans</td>
<td>11.8</td>
<td>8</td>
<td>3</td>
<td>0.02</td>
<td>0.01</td>
<td>6.8</td>
<td>44</td>
</tr>
<tr>
<td>SSCA2</td>
<td>7.0</td>
<td>3</td>
<td>2</td>
<td>0.01</td>
<td>0.03</td>
<td>2.8</td>
<td>12.3</td>
</tr>
</tbody>
</table>

We believe that transactional characteristics of future workloads cannot be determined using just a handful of applications. As concurrent programming and TM methods become popular, workloads should see a swing towards larger, more frequent transactions. With this outlook, applications like yada, labyrinth, bayes, intruder, genome and vacation with transactional residencies of 30% or above appear to be good candidates for comparing different HTM approaches. Amdahl’s law would not justify investment of hardware resources for applications like kmeans and SSCA2 which do not satisfy this criterion. For such applications, the TM approach is still attractive because of the ease of programming it brings. In addition to high transactional residency, moderate to high contention for shared data is also required to highlight performance differences between HTM design points. Hence, in the discussion that follows we suggest the reader to focus on the results for the applications highlighted in Fig. 4.

In our simulations, we have noticed that the application bayes exhibits inconsistent behavior and no clear trends favoring one design point over the other can be inferred. Hence, this application has been excluded from our analysis.

B. Impact of Conflict Arbitration Policy

Fig. 5 shows the impact of varying conflict arbitration policies. Speed-ups on choosing dynamic priorities based on work done over fixed timestamp based priorities have been plotted – plot 5(a) represents the case when requesters abort on conflicts while plot 5(b) corresponds to the case where requesters stall.
The data shows that dynamic priorities work well with all the STAMP benchmarks, especially when used in conjunction with requester-stall conflict rectification. Use of dynamic priorities, especially like the one we have used based on work done, results in improved commit throughput by favoring transactions that have done more work, especially in applications like labyrinth that are characterized by extremely high contention and limited concurrency.

**Figure 5. Impact of conflict arbitration policies**

As transaction sizes and contention increase the impact of this decision becomes more pronounced. This trend can be easily noticed in applications like yada and labyrinth. This strengthens the case for having flexible conflict resolution strategies in HTMs which can adapt to different application characteristics. We believe that TM applications of the future will span a much broader region in Fig. 4.

**C. Impact of Conflict Rectification Policy**

Fig. 6 shows the impact of the choice of conflict rectification policy. Speed-ups on choosing requester-stall over requester-abort have been plotted – plot 6(a) uses fixed priorities whereas plot 6(b) uses dynamic priorities. In general, we see a speed-up, especially in applications like yada and labyrinth, which have large transactions and high contention for shared data. In such applications, significant work can be saved when conflict rectification occur late in requesting transactions. The use of dynamic priorities tends to amplify the observed speed-up.

**Figure 6. Impact of conflict rectification policies**

**D. Impact of LV* configurations**

Fig. 7 shows performance variations seen when running STAMP applications under different LV* configurations. Results compare the percentage speedup for the best performing configuration for each application against the worst performing one. Annotations at the top of each bar identify the best and worst configurations (best/worst). LL-MESI works well for yada while LazEr (dynamic priorities) stall provides significant gains for labyrinth.

Small but noticeable gains can be seen in applications, like SSCA2 and kmeans, that have small transactions (refer to Table 2) when using LazEr with fixed timestamp based priorities without stalls. The transmission of commit and abort messages in such a scenario to release stalls would impose an overhead that is not insignificant when compared to the average transaction lifetime.

Two important observations can be gathered from this data, (a) the best and the worst performing configurations change from application to application, (b) there are considerable performance gains to be had for applications that have high transactional residency and contention. LV* now tracks the best available HTM performance and this, as we have shown in Section II(C), comes at little additional cost in terms of design complexity.
Fig. 7. Performance variations of different LV* configurations

Fig. 8 shows the speedup in transactional performance for different STAMP applications when running on 8 cores as compared to a single core. This data can be used as a basis for comparing our results with other transactional memory proposals.

V. RELATED WORK

The first TM proposals by Herlihy and Moss [9] appeared as early as 1993. More sophisticated HTM proposals like TCC [8] and LogTM [17] have studied the HTM design space based on conflict resolution and version management. Various optimizations and designs have since proposed solutions to deal with issues of scalability in terms of number of cores, memory hierarchy, transaction sizes [1,29] and lifetimes [1,20].

Flexibility and simplicity of design have rarely come together in HTM proposals. Lazy conflict resolution HTM proposals like TCC were limited to a committer-wins policy and required sophisticated commit strategies for large parallel architectures. LTM [1], proposed by Ananian et al., used a requester-wins approach to resolve conflicts between transactions. LogTM [17] employed a requester-stall approach when resolving conflicts, with a software handler to break deadlocks. VTM[20] used age-based priorities while Walilullah et al. [28] adapted a squash-count based priority scheme to avoid starvation in lazy conflict resolution HTMs. Bobba et al. [2] identified performance-degrading interactions between workloads and the underlying TM design. Rigid policy HTMs can fail to adapt to workload characteristics and consequently, some performance gains remain latent.

On the other hand, software and hybrid TM systems have provided great flexibility in the choice of TM policies but, in general, fail to achieve performance comparable to HTMs. A comprehensive study of various conflict resolution techniques has been done by Spear et al. [26] in the context of software transactional memory systems. Recent work by Shriraman et al. [24] studied flexible contention management in software within the framework of hardware-accelerated TM systems. Both studies highlighted the importance of TM policy decisions and suggested that a lazy conflict resolution approach is able to extract good concurrency in several situations. Our experiments with LL-MESI do indeed show this, but not for every application considered in this work.

VI. CONCLUSION AND FUTURE WORK

In this work we have motivated the need and paved a path for the integration of simple yet high-performance flexible TM infrastructure in chip multiprocessors. We have demonstrated a low complexity implementation of two inherently different design points, LazEr and LL-MESI, into one architectural framework called LV*. We have shown that transactional performance offered by different HTM configurations varies over the spectrum of applications we have used and that LV* is capable of tracking the best available HTM performance.

From the data we have gathered from STAMP applications, we notice that as transaction sizes and contention for shared data increases, applications become more sensitive to the choice of conflict resolution policies. Future TM applications are expected to show a much wider range of characteristics and HTM infrastructures would have to be malleable enough to fit their needs. In this context, we feel that models that can characterize applications based on their affinity to conflict resolution policies would prove valuable in setting goals for future multiprocessing systems. The mechanism for priority transport as used by LazEr can easily be used to enable other forms of transaction ordering like those based on number of memory accesses, read/write set sizes, transaction lifetime, priority escalation on aborts and even programmer defined indicators of transaction importance.

REFERENCES


