2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

Transactional Memory Architecture and Implementation for IBM System z

Christian Jacobi, Timothy Slegel
IBM Systems and Technology Group
Poughkeepsie, NY, USA
cjacobi@us.ibm.com, slegel@us.ibm.com

Dan Greiner
IBM Systems and Technology Group
San Jose, CA, USA
dgreiner@us.ibm.com

***Abstract*—We present the introduction of transactional memory into the next generation IBM System z CPU. We first describe the instruction-set architecture features, including requirements for enterprise-class software RAS. We then describe the implementation in the IBM zEnterprise EC12 (zEC12) microprocessor generation, focusing on how transactional memory can be embedded into the existing cache design and multiprocessor shared-memory infrastructure. We explain practical reasons behind our choices. The zEC12 system is available since September 2012.**

I. Introduction And Related Work

Over the last years, the number of CPU cores on a chip and the number of CPU cores connected to a shared memory have grown significantly to support growing workload capacity demand. For example, the IBM zEC12[1] enterprise server [1][2] supports operating system images with up to 101 CPUs. The increasing number of CPUs cooperating to process the same workloads puts significant burden on software scalability; for example, shared queues or data-structures protected by traditional semaphores become hot spots and lead to sub-linear n-way scaling curves. Traditionally this has been countered by implementing finer-grained locking in software, and with lower latency/higher bandwidth interconnects in hardware. Implementing fine-grained locking to improve software scalability can be very complicated and error-prone, and at today's CPU's frequency, the latency of hardware interconnects is limited by the physical dimension of the chips and systems, and by the speed of light.

In [3], Herlihy et al. introduced transactional memory: a group of instructions called a transaction is operating atomically and in isolation (called *serializability* in [3]) on a data structure in memory; the transaction executes optimistically without obtaining a lock, but may need to abort and retry if the operation conflicts with other operations on the same memory locations. The authors propose one implementation with a special transaction cache to hold pending transactional stores.

Various alternative hardware transactional memory designs have been proposed since [3]. In the Transactional Memory Coherence and Consistency (TCC) model [4], all stores performed during a transaction are buffered, and a request to store them out to memory is put on the bus at the end of the transaction. The bus arbitrates between multiple CPUs, and while one CPU is storing its stores, other CPUs are snooping the stores for conflicts and abort their transaction if necessary. A different approach is chosen in LogTM [5]: the transaction speculatively updates the memory but keeps the original value in a log and can restore the original memory content from the log in case of an abort. LogTM allows faster commit than TCC, and the less frequent aborts take longer than in TCC. LogTM uses a directory based eager conflict detection mechanism, where on a local cache miss other CPUs are informed of the transactional access so that they can detect potential conflicts and abort. Recent commercial implementations include Sun Microsystems' Rock [6] and IBM's BlueGene/Q processors [7]; Intel has announced transactional memory for their Haswell CPUs expected in 2013 [8]. A major difference of our architecture is that it supports *constrained transaction* which are guaranteed to eventually succeed.

In [9], architectural semantics for transactional memory have been studied. The authors propose to combine transactional hardware with a software layer that provides 2-phase commit, software handlers for transaction commit/abort, and closed and open nested transactions. In [10], the same group discusses virtualization of transactions to address overflows, interrupts, and other conditions with the help of the operating system. We have chosen to implement a pure hardware transactional system with closed nesting. In our design, each non-constrained transaction needs a fallback path, which then can also be invoked in circumstances where the transaction cannot complete, e.g. due to interrupts or transaction size overflows. Therefore the additional complexity of a software assist layer was not warranted for our implementation. Not relying on a software layer furthermore enables all components of the software stack including firmware, hypervisor, and operating system to exploit transactional memory.

There is a significant body of work on software-based and hybrid hardware/software transactional memory; since this is only remotely related to our work we refer to a transactional memory overview that discusses some of them [11].

The three use cases we considered during the definition of transactional memory are lock elision, lock-free data structures, and general code optimization. In lock elision [12], a data structure that is typically guarded by a lock is accessed with transactional memory operations without first obtaining the lock. If the transaction aborts due to conflicts with other CPUs, the program can obtain the lock as a fallback path (see figure 1). Every transaction must check that the lock is free to prevent concurrent operation of a transactional CPU and a CPU currently in the fallback path. This method also works with programs that have only been partially changed to use transactional memory, which is important for realistic introduction of transactional memory into large software products. In [12], lock elision is defined with special instructions; we embed the idea into the general transactional memory context. Lock free data structures have been studied extensively (see e.g. [13]). Transaction semantics provide a more powerful and easy-

[1]IBM, System z, z/OS, z/Architecture, zEnterprise, Blue Gene are trademarks of the International Business Machines Corporation.

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.12

```
        LHI      R0,0            *initialize retry count=0
loop    TBEGIN                   *begin transaction
        JNZ      abort           *go to abort code if CC!=0
        LT       R1,lock         *load&test the fallback lock
        JNZ      lckbzy          *branch if lock busy
        ...perform operation...
        TEND                     *end transaction
        ...
lckbzy  TABORT                   *abort if lock busy; this resumes after TBEGIN

abort   JO       fallback        *no retry if CC=3
        AHI      R0,1            *increment retry count
        CIJNL    R0,6,fallback   *give up after 6 attempts
        PPA      R0,TX           *random delay based on retry count
        ... potentially wait for lock to become free
        J        loop            *jump back to retry

fallback
        OBTAIN   lock            *using Compare&Swap
        ...perform operation...
        RELEASE  lock
        ...
```

Figure 1: Example Transaction

to-use foundation than for example a simple compare-and-swap instruction. Lastly, transactions provide a mechanism for code optimization, for example re-ordering code more aggressively, relying on the atomicity and isolation of memory accesses and the provided register-rollback for correctness [14].

During the definition of the transactional memory facility, it became clear very early that special care had to be taken for software testability and debug. Transactional memory poses interesting challenges [15]: for example, the non-transactional fallback path is rarely exercised which may lead to test coverage problems; when a program fails for example with an access exception inside a transaction, the memory and register state is rolled back due to transaction abort, which makes post-mortem analysis more difficult; and lastly, in our implementation interactive debugging is restricted by the fact that interrupts cause transactions to abort, and so for example, setting break-points inside a transaction would be impractical. We discussed many alternative solutions with IBM software teams in order to minimize hardware cost and complexity while meeting the requirements for efficient software development. In section 2, we describe the instruction set architecture for transactional memory, including features for software test and debug.

The design of a high-speed, scalable, and reliable multiprocessor shared memory protocol is very complex and time-consuming and represents a significant investment. The microprocessor design also evolves from generation to generation without a complete redesign. It was therefore clear from the beginning that transactional memory support must fit into a mostly unchanged SMP protocol, and that changes to the microprocessor core design should be minimized. At the same time the design must provide robust transactional memory performance to compete with the performance of obtaining idle locks; otherwise a compiler could not know at compile time whether to replace a lock with a transaction. We describe our implementation in section 3. In section 4 we evaluate the performance of the transactional memory implementation under a set of micro-benchmarks. The results show good scalability even for very high numbers of CPUs under realistic contention scenarios. We also provide some early results on real-world code. Section 5 summarizes the paper.

II. Instruction Set Architecture

A. New Instructions and General TX Operation

The Transactional Execution (TX) Facility provides 6 new instructions to the z/Architecture [16], as well as a few new control bits. This section gives an overview of the central features of the facility, further details can be found in [16].

Transactions are formed by pairs of *Transaction Begin* and *Transaction End* instructions (TBEGIN and TEND, respectively). Except as described below, either all or none of the instructions inside a transaction are executed (*atomicity*), and all operand accesses to memory are performed isolated (sometimes called serializable, block-concurrent, or also atomic), that is, other CPUs and the I/O subsystem cannot observe changes made by the transaction before it successfully ends, and the transaction cannot observe changes made by other CPUs or the I/O subsystem during the transaction.

Transactions may abort. There are various reasons for transactions to abort, including interrupts (e.g., page faults, divide by zero, or asynchronous interrupts like timer and I/O), exceeding the maximum nesting depth, overflow of the CPU's capability to track transactional accesses to memory (*footprint overflow*), or conflicts on accessed memory locations with other CPUs or the I/O subsystem that would cause an isolation violation if the transactional execution would continue. Privileged instructions that modify the control state of the CPU and some other complex instructions are not allowed inside a transaction and always lead to a transaction abort.

The architecture requires that the partial execution of the transaction before an abort was detected is isolated with respect to other CPUs and I/O (this is referred to as *opacity* in [17]). This

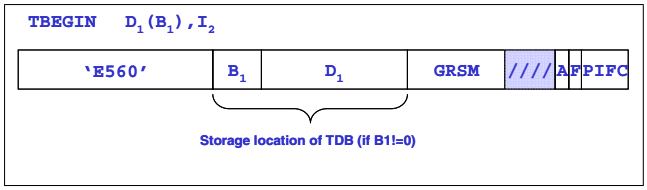

Figure 2: TBEGIN instruction text

was an important request from our software community since it prevents execution based on inconsistent data. For example, one transaction might pop the last element from a stack by updating the count of elements to 0 and by setting the stack pointer to NULL. If another transaction would first read the old non-zero element count, and then would proceed to access the stack pointer without NULL-checking, a page fault would result. Such situations are prevented by requiring that the entire execution is isolated even if the transaction aborts. Herlihy [3] proposed a *validate* instruction to prevent such *zombie transactions* [18]. We choose the stronger isolation since it is easier to use for software developers and does not require additional path length for the intermediate validations.

The architecture also requires isolation of transactions against non-transactional accesses done by other CPUs and I/O (*strong atomicity* in [19]). Again this was an important requirement from our software community to be able to mix transactional and traditional locking-based code in order to ease stepwise introduction of TM technology into existing programs.

Execution of TBEGIN sets the Condition Code (CC) to 0. If a transaction aborts, the *Instruction Address* in the *Program Status Word* (PSW) is restored to the instruction immediately after the TBEGIN, and a condition code is set to a non-zero value. A typical program will test the condition code after TBEGIN to either start the actual transaction processing (CC=0) or branch to the *abort handler* (CC!=0). The abort handler may return to retry the transaction, or it may perform the same functionality non-transactionally on a *fallback path*. An example is shown in figure 1.

Depending on whether the CPU considers the abort reason transient (e.g. another CPU conflicting) or permanent (e.g. a restricted instruction), the condition code is either set to 2 or 3. This allows the program a quick check on whether it should retry the transaction (with a threshold count) or immediately branch to a non-transactional fallback path. Depending on the program, certain clean-up is necessary before repeating the transaction, like restoring certain registers (see below).

Before repeating a transaction after a transient abort, it often makes sense to introduce a random delay that increases with the number of aborts, in order to prevent harmonic, repetitive, aborts due to conflicts between two CPUs (for example using random exponential back-off). The optimal delay distribution may depend on the particular abort reason, specifics of the design of the CPU generation, and details of the SMP configuration. In order to avoid changing the program to adjust the delays to these design parameters, the new *Perform Processor TX-Abort Assist* (PPA with function code TX) instruction is introduced. The program passes the current abort count to the instruction, which then performs a random delay optimal for the current configuration; that way software does not have to be adjusted for future machine generations or different configurations.

If the CPU is already in transactional execution mode when a TBEGIN is executed, a nested inner transaction is started, and the CPU increments the transaction nesting depth. The maximum supported nesting depth is 16. The TEND instruction closes a transaction by decrementing the current nesting depth; if the current nesting depth is 1, execution of TEND commits the transaction and the CPU leaves transactional execution mode. If a transaction abort happens on a nested transaction, the entire nest of transactions is aborted (*flattened nesting*), the nesting depth is set to 0, and execution continues at the instruction after the outermost TBEGIN.

The *Extract Transaction Nesting Depth* (ETND) instruction can be used to load the current nesting depth into a General Register (GR).

The support of nested transactions is important to software in certain cases. Compilers may produce code that calls sub-routines from within transactions, and if those sub-routines were themselves compiled to use transactions, nesting occurs. Sometimes it is important for a sub-routine to know whether it is being called transactionally, which can be done quickly with ETND by checking whether the depth is 0.

While the CPU is in transactional mode, stores performed by the CPU are not made visible to other CPUs or the I/O subsystem until the outermost TEND completes. If the transaction aborts, all stores done during the transaction are discarded. The exception are stores performed by the *Non-transactional Store* (NTSTG) instruction; these 8-byte stores are also isolated, that is, not visible to other CPUs and I/O while the transaction executes, but unlike normal stores they are committed to memory even in the case of transaction abort. The main use case for NTSTG is transactional debugging: a programmer/compiler can store intermediate results into memory and analyze the data even in the abort case to see which program path and data was observed prior to the abort (*breadcrumb debugging*). The architecture requires that the memory locations stored to by NTSTG do not overlap with other stores from the transaction.

The *Transaction Abort* (TABORT) instruction causes an immediate abort. The operand provides an abort reason code which is placed into the optional Transaction Diagnostic Block (see below). The least significant bit of the abort code determines whether the condition code is set to 2 or 3 to indicate transient versus permanent abort to the abort handler.

B. TBEGIN Control Fields

The TBEGIN instruction has a set of operand fields (see figure 2). The *General Register Save Mask* (GRSM) is an 8 bit field, each bit corresponding to an even/odd pair of the 16 General Registers (GRs). At the execution of the outermost TBEGIN, the pairs indicated with a '1' in the mask are saved, and are restored to their pre-TBEGIN content in the case of transaction abort. GR-pairs not indicated in the mask keep their current value in case of abort (this is an exception to the "all or nothing" atomicity rule, since modified state survives the abort). Saving only a subset of GRs during TBEGIN speeds up execution, and not restoring all GRs during abort provides information for debugging and analysis.

The IBM z/Architecture supports additional register sets, namely Access Registers (ARs) and Floating-Point Registers (FPRs), for which no save/restore mechanism is provided - it is up to software

to save necessary registers before entering a transaction, and to restore those registers in the abort handler.

Some software may expect ARs or FPRs to not be modified and thus does not provide save/restore in the abort handler. But when calling a sub-routine — potentially in a linked library — the sub-routine might inadvertently change one of those registers. Even worse, a sub-routine might for example use an AR as scratch space and later restore it. This is not observable if the transaction does not abort, but in the (rare) case of an abort while the register is modified, unpredictable results may occur due to modified register content after the abort. Such failures may be rare and extremely difficult to debug. To protect against such situations, the TBEGIN instruction provides AR and FPR modification control bits. Any instruction attempting to modify an AR or FPR, respectively, leads to a restricted-instruction abort if the respective control bit is '0'. For nested transactions, the effective control is the 'AND' of all control bits in the nest.

C. Interruption Filtering

When a program-exception condition is detected during normal program execution, an interruption into the operating system occurs. The PSW at which the exception is detected is stored as *program-old PSW*, and a *program-new PSW* pointing to the OS interrupt handler is loaded. The OS can then service the interrupt (e.g. page in memory from disk) and then return to program execution by loading the saved program-old PSW into the PSW. (The same concept is used for other interruptions like I/O, but we omit the details here.)

The transactional memory architecture provides control over whether certain exceptions detected during a transaction actually lead to a program interruption into the OS, or whether the interruption is *filtered*. In both cases the transaction is first aborted. If an interruption into the OS occurs, the program-old PSW will point to the instruction after the TBEGIN with a non-zero condition code. That way, when the OS returns control to the program, the program knows to execute the abort handler before potentially retrying the transaction. In the case of a filtered interruption, the condition code is also non-zero, and the program continues execution after the outermost TBEGIN without first trapping into the OS.

Exceptions are categorized into 4 groups. First, some exceptions can never occur during transactional execution, for example since they are related to specific instructions that are restricted in transactions. Second is a group of exceptions that is always considered a programming error and thus is always causing interruption into the OS; examples include undefined instruction op-codes in the instruction stream. Third is a group of exceptions related to accessing of memory, for example page fault exceptions. Fourth is a group of exceptions related to arithmetic and data, e.g. divide-by-zero or overflow exceptions. The third and fourth group of exceptions can be filtered under the control of the *Program Interruption Filtering Control* (PIFC) field of the TBEGIN instruction. Values 0 to 2 in this field correspond to no filtering, filtering of group 4 only, and filtering of groups 3 and 4, respectively. In a nested transaction, the effective PIFC is the highest value of all TBEGINs in the nest.

Interruption filtering is useful in many speculative program optimizations, e.g. by not performing null-pointer checks before accessing data, or not performing NaN or zero checks before performing computations. Instead these (depending on the program)

```
TBEGINC   *begin constrained transaction
...perform operation...
TEND      *end transaction
...
```

Figure 3: Constrained transaction example

rare conditions can be treated in the transaction abort handler, improving performance for the normal case and penalizing the rare case only. Of course null-pointer or NaN checking are only examples; once the infrastructure is available, the compiler can perform general if-then-else speculation and other optimization using the same concepts [14].

It is important for the program to adhere to certain rules when using interruption filtering. For example, a filtered page fault encountered during a transaction is not reported to the OS; if the abort handler does not access the same memory locations non-transactionally, the program may never trap the page fault into the OS, the page fault will never be resolved, and thus the transaction continues to fail every time it is executed. Exceptions related to instruction fetching are never filtered. The program-old PSW will indicate a transient abort condition code of 2, so that the program usually repeats the transaction immediately after the OS handled the interrupt. If instruction fetching exceptions were filtered, a page fault on an instruction page that is only used during transactional execution would never be resolved by the OS and always cause that transaction code to abort.

D. Constrained Transactions

Transactions started with TBEGIN are not assured to ever successfully complete with TEND, since they can experience an aborting condition at every attempted execution, e.g. due to repeating conflicts with other CPUs. This requires that the program supports a fallback path to perform the same operation non-transactionally, e.g. by using traditional locking schemes. This puts significant burden on the programming and software verification teams, especially where the fallback path is not automatically generated by a reliable compiler.

Many transactions operating on shared data structures are expected to be short, touch only few distinct memory locations, and use simple instructions only. For those transactions, the concept of *constrained transactions* is available; under normal conditions, the CPU assures that constrained transactions eventually end successfully, albeit without giving a strict limit on the number of necessary retries. A constrained transaction starts with a TBEGINC instruction and ends with a regular TEND. Implementing a task as constrained or non-constrained transaction typically results in very comparable performance, but constrained transactions simplify software development by removing the need for a fallback path.

A transaction initiated with TBEGINC must follow a list of programming constraints; otherwise the program takes a non-filterable constraint-violation interruption. The constraints include: the transaction can execute a maximum of 32 instructions, all instruction text must be within 256 consecutive bytes of memory; the transaction contains only forward-pointing relative branches (hence no loops or sub-routine calls); the transaction can access a maximum of 4 aligned octowords (32 bytes) of memory; and restriction of the instruction-set to exclude complex instructions

like decimal or floating-point operations. The constraints are chosen such that many common operations like double-linked list-insert/delete operations can be performed, including the very powerful concept of atomic compare-and-swap targeting up to 4 aligned octowords. At the same time the constraints were chosen conservatively such that future CPU implementations can assure transaction success without needing to adjust the constraints, since that would otherwise lead to software incompatibility.

TBEGINC mostly behaves like TBEGIN, except that the FPR control and the program interruption filtering fields do not exist and the controls are considered to be zero. On a transaction abort, the instruction address is set back directly to the TBEGINC instead to the instruction after, reflecting the immediate retry and absence of an abort path for constrained transactions.

Nested transactions are not allowed within constrained transactions, but if a TBEGINC occurs within a non-constrained transaction it is treated as opening a new non-constrained nesting level just like TBEGIN would. This can occur e.g. if a non-constrained transaction calls a sub-routine that uses a constrained transaction internally.

Since interruption filtering is implicitly off, all exceptions during a constrained transaction lead to an interruption into the OS. Eventual successful finishing of the transaction of course relies on the capability of the OS to page-in the at most 4 pages touched by any constrained transaction. The OS must also ensure time-slices long enough to allow the transaction to complete.

Figure 3 shows the constrained-transactional implementation of the code in figure 1, assuming that the constrained transactions does not interact with other locking-based code. No lock testing is shown therefore, but could, of course, be added if constrained transactions and lock-based code were mixed.

E. Debugging Features

Reliable software is essential for enterprise class computing, and transactional memory poses interesting challenges to how software debugging and testing is performed during the software development cycle and during field failure analysis. Significant effort was spent on the development of architectural features to support debugging and testing.

1) Transaction Diagnostic Block: The TBEGIN instruction has an optional address operand called the *Transaction Diagnostic Block (TDB) Address*. The TDB is not used during normal transaction processing, but if a transaction aborts and a TDB Address is specified on the outermost TBEGIN, detailed information about the abort is stored in the TDB. The TDB is 256 bytes in length, and its fields include: (i) Transaction Abort Code, indicating the detailed reason for the abort; (ii) Conflict Token, providing the address that caused a conflict with another CPU; this field cannot always be provided and there is a bit indicating the validity; (iii) Aborted-Transaction Instruction Address, indicating the IA at which the abort was detected; (iv) Exception information like Program Interruption Code and Translation Exception Address; (v) the content of all GRs at the time of abort; and (vi) CPU specific information not formally architected. The last item provides detailed CPU-generation dependent information on the details of why the transaction aborted, and which path the program took from the outermost TBEGIN to the abort IA.

It is expected that extracting the information and storing the TDB on transaction abort takes a number of CPU cycles, and thus only code in debug/test or with extremely low abort rates will enable TDBs on performance-sensitive transactions. During initial hardware validation of the transactional facility, the information in the TDB was invaluable for debugging test program and hardware/firmware problems. We expect similar usefulness for debugging application code.

A second copy of the TDB is stored into the processor prefix area (a memory area containing reserved locations specific for each CPU in the system) on every abort due to a program interruption; this is valuable for post-mortem failure analysis after a program ended abnormally, e.g. on an access exception.

2) PER: In z/Architecture, traditionally Program Event Recording provides a hardware mechanism to trigger a program interruption for certain events. The supported events include stores into a specified memory range, execution of instructions from a specified memory range, and branching into a specified memory range. This mechanism is used extensively for software debugging, for example in z/OS SLIP traps, or in GDB under Linux for setting break- or watch-points.

Detection of a PER event inside a transaction causes a transaction abort and a non-filterable interruption into the OS. Two new features are added to PER for transactional memory: (i) *PER Event Suppression* suppresses any PER event while running in transactional mode, and (ii) the new *PER TEND event* triggers on successful execution of an outermost TEND instruction.

For example, if a debugger is running in single instruction mode, a PER instruction-fetch event is enabled for the entire address range. PER event suppression can be used to avoid aborting every single transaction on the first instruction after the TBEGIN. This effectively makes entire transactions look like single ”big instructions” in the single-step mode.

Another use case of PER is monitoring for stores into a specific memory range for implementing watch-points. Without event suppression, a transaction modifying memory in the monitored range always aborts and eventually takes the fallback path. To enable debugging of the transactional code itself, event suppression can be enabled alongside the new PER TEND-event, which triggers at the ending of every transaction. The debugger can then check all active watch-points for whether the memory content changed and enter the interactive debugging mode in that case.

For constrained transactions, it is up to the OS to enable event suppression after a PER event caused a transaction abort, in order to enable the transaction to complete on the next retry. The OS can use PER TEND to disable event suppression after successful transaction completion.

3) Transaction Diagnostic Control: Most transactions will abort only infrequently, and the point of abort inside the transaction may be non-uniformly distributed. For example, certain instructions cause conflicts with other CPUs more frequently than other instructions. This creates unique debugging and testing challenges. The abort path and fallback path might be sparsely exercised leading to poor testing coverage. Also, the random distribution of the abort point may lead to unusual corner cases after the abort if residual state survives the abort (for example non-restored registers). This may lead to program failures that are very hard to reproduce and to debug.

In order to enhance the testing coverage of the abort path and to protect against untested corner cases, the Transaction Diagnostic Control is provided to force random aborts. At one setting, the CPU is instructed to often, randomly abort transactions at a random point. At a more aggressive setting, the CPU is instructed to abort every transaction at a random point but at latest before the outermost TEND instruction. The latter setting can be used to stress the reaching of the retry-threshold and force the non-transactional fallback path to be used. This more aggressive setting is treated like the less aggressive setting for constrained transactions. The Transaction Diagnostic Control can be enabled by the OS for testing specific programs.

III. Implementation

The main implementation components of the transactional memory facility are a transaction-backup register file for holding pre-transaction GR content, a cache directory to track the cache lines accessed during the transaction, a store cache to buffer stores until the transaction ends, and firmware routines to perform various complex functions. In this section we describe the detailed implementation.

A. System Background

The transactional execution facility is first implemented in the IBM zEC12 processor [1], the successor of the z196 processor described in [20]. The processor can decode 3 instructions per clock cycle; simple instructions are dispatched as single micro-ops, and more complex instructions are cracked into multiple micro-ops. The micro-ops are written into a unified issue queue, from where they can be issued out-of-order. Up to two fixed-point, one floating-point, two load/store, and two branch instructions can execute every cycle. A Global Completion Table (GCT) holds every micro-op. The GCT is written in-order at decode time, tracks the execution status of each micro-op, and completes instructions when all micro-ops of the oldest instruction group have successfully executed.

The L1 data cache is a 96KB 6-way associative cache with 256 byte cache-lines and 4 cycle use-latency, coupled to a private 1MB 8-way associative 2nd-level data cache with 7 cycles use-latency penalty for L1 misses. Both L1 and L2 caches are store-through. Six cores on each *CP chip* share a 48MB 3rd-level store-in cache, and six CP chips are connected to an off-chip 384MB 4th-level cache, packaged together on a glass-ceramic multi-chip module (MCM). Up to 4 MCMs can be connected to a coherent SMP system with up to 144 cores (not all cores are available to run customer workload).

Coherency is managed with a variant of the MESI protocol. Cache-lines can be owned read-only (shared) or exclusive; the L1 and L2 are store-through and thus do not contain dirty lines. The L3 and L4 caches are store-in and track dirty states. Each cache is inclusive of all its connected lower level caches.

Coherency requests are called *cross interrogates* (XI) and are sent hierarchically from higher-level to lower-level caches, and between the L4s. When one core misses the L1 and L2 and requests the cache line from its local L3, the L3 checks whether it owns the line, and if necessary sends an XI to the currently owning L2/L1 under that L3 to ensure coherency, before it returns the cache line to the requestor. If the request also misses the L3, the L3 sends a request to the L4 which enforces coherency by sending XIs to all necessary L3s under that L4, and to the neighboring L4s. Then the L4 responds to the requesting L3 which forwards the response to the L2/L1.

Note that due to the inclusivity rule of the cache hierarchy, sometimes cache lines are XI'ed from lower-level caches due to evictions on higher-level caches caused by associativity overflows from requests to other cache lines. We call those XIs *LRU XIs*.

Demote-XIs transition cache-ownership from exclusive into read-only state, and Exclusive-XIs transition cache-ownership from exclusive into invalid state. Demote- and Exclusive-XIs need a response back to the XI sender. The target cache can accept the XI, or send a reject response if it first needs to evict dirty data before accepting the XI. The L1/L2 are store through, but may reject demote- and exclusive XIs if they have stores in their store queues that need to be sent to L3 before downgrading the exclusive state. A rejected XI will be repeated by the sender. Read-only-XIs are sent to caches that own the line read-only; no response is needed for such XIs since they cannot be rejected. The details of the SMP protocol are very similar to those described for the IBM z10 in [21].

B. Transactional Instruction Execution

The instruction decode unit (IDU) keeps track of the current transaction nesting depth (TND, see figure 4). When the IDU receives a TBEGIN instruction, the nesting depth is incremented, and conversely decremented on TEND instructions. The nesting depth is written into the GCT for every dispatched instruction. When a TBEGIN or TEND is decoded on a speculative path that later gets flushed, the IDU's nesting depth is refreshed from the youngest GCT entry that is not flushed. The transactional state is also written into the issue queue for consumption by the execution units, mostly by the Load/Store Unit (LSU).

Similar to the nesting depth, the IDU/GCT collaboratively track the AR/FPR-modification masks through the transaction nest; the IDU can place an abort-request into the GCT when an AR/FPR-modifying instruction is decoded and the modification mask blocks that. When the instruction becomes next-to-complete, completion is blocked and the transaction aborts. Other restricted instructions are handled similarly, including TBEGIN if decoded while in a constrained transaction, or exceeding the maximum nesting depth.

An outermost TBEGIN is cracked into multiple micro-ops depending on the GR-Save-Mask; each micro-op will be executed by one of the two FXUs to save a pair of GRs into a special transaction-backup register file, that is used to later restore the GR content is case of a transaction abort. Also the TBEGIN spawns micro-ops to perform an accessibility-test for the TDB if one is specified; the address is saved in a special purpose register for later usage in the abort case. At the decoding of an outermost TBEGIN, the instruction address and the instruction text of the TBEGIN are also saved in special purpose registers for a potential abort processing later on.

TEND and NTSTG are single micro-op instructions; NTSTG is handled like a normal store except that it is marked as non-transactional in the issue queue so that the LSU can treat it appropriately. TEND is a no-op at execution time, the ending of the transaction is performed when TEND completes.

As mentioned, instructions that are within a transaction are marked as such in the issue queue, but otherwise execute mostly

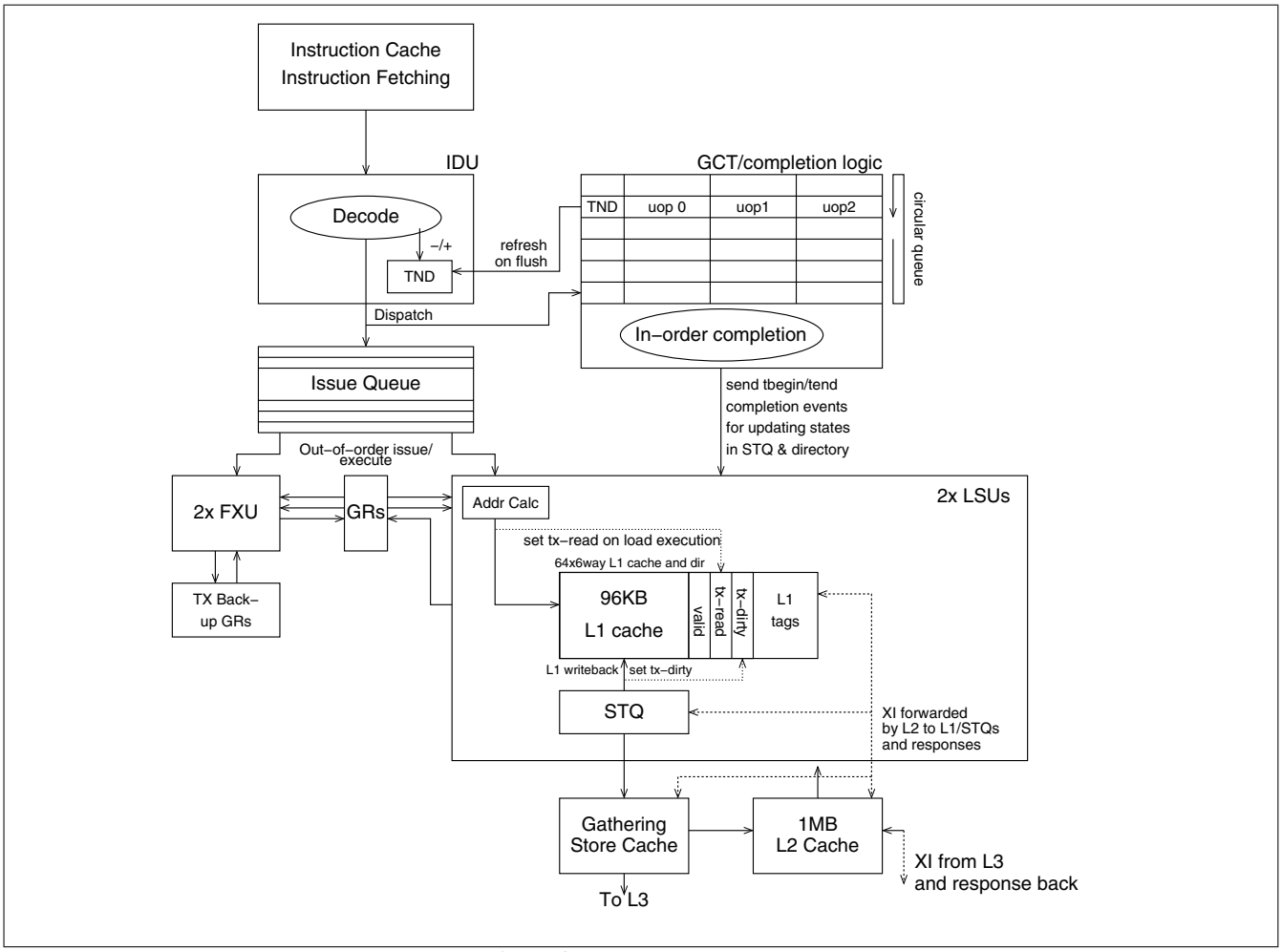

Figure 4: Block diagram of CPU

unchanged; the LSU performs isolation tracking as described in the next section.

Since decoding is in-order, and since the IDU keeps track of the current transactional state and writes it into the issue queue along with every instruction from the transaction, execution of TBEGIN, TEND, and instructions before, within, and after the transaction can be performed out-of-order. It is even possible (though unlikely) that TEND is executed first, then the entire transaction, and lastly the TBEGIN executes. Of course program order is restored through the GCT at completion time. The length of transactions is not limited by the size of the GCT, since GRs can be restored from the backup register file.

During execution, the PER events are filtered based on the Event Suppression Control, and a PER TEND event is detected if enabled. Similarly, while in transactional mode, a pseudo-random generator may be causing the random aborts as enabled by the Transaction Diagnostics Control.

C. Tracking for Transactional Isolation

The Load/Store Unit tracks cache lines that were accessed during transactional execution, and triggers an abort if an XI from another CPU (or an LRU-XI) conflicts with the footprint. If the conflicting XI is an exclusive or demote XI, the LSU rejects the XI back to the L3 in the hope of finishing the transaction before the L3 repeats the XI. This *stiff-arming* is very efficient in highly contended transactions. In order to prevent hangs when two CPUs stiff-arm each other, a XI-reject counter is implemented, which triggers a transaction abort when a threshold is met.

The L1 cache directory is traditionally implemented with SRAMs. For the transactional memory implementation, the valid bits (64 rows x 6 ways) of the directory have been moved into normal logic latches, and are supplemented with two more bits per cache line: the tx-read and tx-dirty bits.

The tx-read bits are reset when a new outermost TBEGIN is decoded (which is interlocked against a prior still pending transaction). The tx-read bit is set at execution time by every load instruction that is marked transactional in the issue queue. Note that this can lead to over-marking if speculative loads are executed, for example on a mispredicted branch path. The alternative of setting the tx-read bit at load completion time was too expensive for silicon area, since multiple loads can complete at the same time, requiring many read-ports on the load-queue.

Stores execute the same way as in non-transactional mode, but a transaction mark is placed in the store queue (STQ) entry of the store instruction. At writeback time, when the data from the STQ is written into the L1, the tx-dirty bit in the L1-directory is set for the written cache line. Store writeback into the L1 occurs only after the store instruction has completed, and at most one store is written back per cycle. Before completion and writeback, loads can access the data from the STQ by means of store-forwarding; after write-back, the CPU can access the speculatively updated data in the L1. If the transaction ends successfully, the tx-dirty bits of all cache-lines are cleared, and also the tx-marks of not yet written stores are cleared in the STQ, effectively turning the pending stores into normal stores.

On a transaction abort, all pending transactional stores are invalidated from the STQ, even those already completed. All cache lines that were modified by the transaction in the L1, that is, have the tx-dirty bit on, have their valid bits turned off, effectively removing them from the L1 cache instantaneously.

As described in Section 2, the architecture requires that before completing a new instruction we ensure that isolation of the transaction read- and write-set is maintained. This is ensured by stalling instruction completion at appropriate times when XIs are pending; we allow speculative out-of-order execution, optimistically assuming that the pending XIs are to different addresses and not actually cause a transaction conflict. This design fits very naturally with the XI-vs-completion interlocks that are implemented on prior systems to ensure the strong memory ordering that the architecture requires [22].

When the L1 receives an XI, it accesses the directory to check validity of the XI'ed address in the L1, and if the tx-read bit is active on the XI'ed line and the XI is not rejected, the LSU triggers an abort. When a cache line with active tx-read bit is LRU'ed from the L1, a special LRU-extension vector remembers for each of the 64 rows of the L1 that a tx-read line existed on that row. Since no precise address tracking exists for the LRU extensions, any non-rejected XI that hits a valid extension row the LSU triggers an abort. Providing the LRU-extension effectively increases the read footprint capability from the L1-size to the L2-size and associativity, provided no conflicts with other CPUs against the non-precise LRU-extension tracking causes aborts; section 4 contains statistical analysis of the effectiveness of the LRU extension.

The store footprint is limited by the store cache size (next section) and thus implicitly by the L2 size and associativity. No LRU-extension action needs to be performed when a tx-dirty cache line is LRU'ed from the L1.

D. Store Cache

In prior systems, since the L1 and L2 are store-through caches, every store instruction causes an L3 store access; with now 6 cores per L3 and further improved performance of each core, the store rate for the L3 (and to a lesser extent for the L2) becomes problematic for certain workloads. In order to avoid store queuing delays a gathering store cache had to be added, that combines stores to neighboring addresses before sending them to the L3.

For transactional memory performance, it is acceptable to kill every tx-dirty cache line from the L1 on transaction aborts, because the L2 cache is very close (7 cycles L1 miss penalty) to bring back the clean lines. It would however be unacceptable for performance (and silicon area for tracking) to have transactional stores write the L2 before the transaction ends and then kill all dirty L2 cache lines on abort (or even worse on the shared L3).

The two problems of store bandwidth and transactional memory store handling can both be addressed with the gathering store cache. The cache is a circular queue of 64 entries, each entry holding 128 bytes of data with byte-precise valid bits. In non-transactional operation, when a store is received from the LSU, the store cache checks whether an entry exists for the same address, and if so gathers the new store into the existing entry. If no entry exists, a new entry is written into the queue, and if the number of free entries falls under a threshold, the oldest entries are written back to the L2 and L3 caches.

When a new outermost transaction begins, all existing entries in the store cache are marked *closed* so that no new stores can be gathered into them, and eviction of those entries to L2 and L3 is started. From that point on, the transactional stores coming out of the LSU STQ allocate new entries, or gather into existing transactional entries. The writeback of those stores into L2 and L3 is blocked, until the transaction ends successfully; at that point subsequent (post-transaction) stores can continue to gather into existing entries, until the next transaction closes those entries again.

The store cache is queried on every exclusive or demote XI, and causes an XI reject if the XI compares to any active entry. If the core is not completing further instructions while continuously rejecting XIs, the transaction is aborted at a certain threshold to avoid hangs.

The LSU requests a transaction abort when the store cache overflows. The LSU detects this condition when it tries to send a new store that cannot merge into an existing entry, and the entire store cache is filled with stores from the current transaction. The store cache is managed as a subset of the L2: while transactionally dirty lines can be evicted from the L1, they have to stay resident in the L2 throughout the transaction. The maximum store footprint is thus limited to the store cache size of 64 x 128 bytes, and it is also limited by the associativity of the L2. Since the L2 is 8-way associative and has 512 rows, it is typically large enough to not cause transaction aborts.

If a transaction aborts, the store cache is notified and all entries holding transactional data are invalidated. The store cache also has a mark per doubleword (8 bytes) whether the entry was written by a NTSTG instruction - those doublewords stay valid across transaction aborts.

E. Millicode-implemented functions

Traditionally, IBM mainframe server processors contain a layer of firmware called millicode which performs complex functions like certain CISC instructions, interruption handling, system synchronization, and RAS. Firmware resides in a restricted area of main memory that customer programs cannot access. When hardware detects a situation that needs to invoke millicode, the instruction fetching unit switches into *millicode mode* and starts fetching at the appropriate location in the millicode memory area.

For transactional memory, millicode is involved in various complex situations. Every transaction abort invokes a dedicated millicode sub-routine to perform the necessary abort steps. The

transaction-abort millicode starts by reading special-purpose registers (SPRs) holding the hardware-internal abort reason, potential exception reasons, and the aborted instruction address, which millicode then uses to store a TDB if one is specified. The TBEGIN instruction text is loaded from an SPR to obtain the GR-save-mask, which is needed for millicode to know which GRs to restore. The CPU supports a special millicode-only instruction to read out the backup-GRs and copy them into the main GRs. The TBEGIN instruction address is also loaded from an SPR to set the new instruction address in the PSW to continue execution after the TBEGIN once the millicode abort sub-routine finishes. That PSW may later be saved as program-old PSW in case the abort is caused by a non-filtered program interruption.

The TABORT instruction is millicode implemented; when the IDU decodes TABORT, it instructs the instruction fetch unit to branch into TABORT's millicode, from which millicode branches into the common abort sub-routine.

The Extract Transaction Nesting Depth (ETND) instruction is also millicoded, since it is not performance critical; millicode loads the current nesting depth out of a special hardware register and places it into a GR.

The PPA instruction is millicoded; it performs the optimal delay based on the current abort count provided by software as an operand to PPA, and also based on other hardware internal state.

For constrained transactions, millicode keeps track of the number of aborts. The counter is reset to 0 on successful TEND completion, or if an interruption into the OS occurs (since it is not known if or when the OS will return to the program). Depending on the current abort count, millicode can invoke certain mechanisms to improve the chance of success for the subsequent transaction retry. The mechanisms involve, for example, successively increasing random delays between retries, and reducing the amount of speculative execution to avoid encountering aborts caused by speculative accesses to data that the transaction is not actually using. As a last resort, millicode can broadcast to other CPUs to stop all conflicting work, retry the local transaction, before releasing the other CPUs to continue normal processing. Multiple CPUs must be coordinated to not cause deadlocks, so some serialization between millicode instances on different CPUs is required.

IV. Performance Evaluation

We conducted a set of experiments to measure the performance of transactional memory in comparison to lock-based concurrency, over a range of realistic and artificial conditions. We used micro-benchmarks for these experiments since application-level transactional memory exploitation is still in development. The benchmarks use different pools of shared variables ranging from a single variable to 10k variables, each on a separate cache line. Each CPU repeatedly picks either 1 or 4 random variables from the pool and increments the chosen variable(s). If the pool consists of only 1 variable, we use 4 consecutive cache lines for the tests that update 4 variables.

We use both coarse and fine grained locking for comparison with transactional memory. For coarse-grained locking, we use a single lock for the entire pool. For fine-grained locking, we define a lock for each variable, each lock sitting on a separate cache line. In both cases we use a simple mutex algorithm, which first tests the lock to be empty and spins if necessary, then uses compare-and-swap to set the lock, which starts over if not successful; the unlock uses a simple store to unset the lock.

For non-constrained transactions, we use the code from figure 1; for the fallback lock we use the single coarse-grained lock in all experiments. The constrained transaction code from figure 3 does not need fallback locks.

Each CPU independently picks random variables and performs the incrementing on the shared variables. We use the *Store Clock Fast* instruction to measure the time between each lock/tbegin and unlock/tend, but exclude the overhead such as random number generation from the results. The overhead is significant for small numbers of CPUs since the path length for lock/update/unlock is very short compared to computing four random variables. From the measured times we compute the system throughput as the quotient of the number of CPUs divided by the average time per update. All results are normalized to a throughput of 100 for 2 CPUs concurrently updating a single variable from a pool of 1 variable.

Contention for most objects is relatively low in typical commercial applications, and if an object is accessed once it likely will be accessed again by the same CPU. As a result, many lock obtain/release operations are performed with L1-cache hits and thus are very fast. It was important in the design of transactional memory that starting and ending a transaction has similar overhead as locking and releasing a lock that is in the L1-cache; otherwise any performance gain from better behavior on contended locks could have been eroded by non-contended locks. Our experiments cover this case by having only a single CPU participate, and by setting the pool size to a single cache line. In that experiment, transactions outperform locks by 30%. This is mostly due to the longer path length of the lock and release code. The results also show that the overhead of testing the lock in the non-constrained transaction (see figure 1) is insignificant since the branch is perfectly predictable in this case; the performance difference between constrained and non-constrained transactions is 0.4%.

One major drawback of lock-based methods is the complexity involved with fine-grained locking. For example, in the case of updating 4 random variables, the programmer would have to ensure that locks are acquired in a certain order to prevent deadlocks, which in practice can be very hard, e.g. when the objects involved in an operation are not all known a priori. Thus one major use case of transactions is to allow fine grained concurrency in cases where fine-grained locking is hard to achieve. Figure 5(a) shows the performance of updating 4 random variables from a large pool (1k and 10k entries), using transactions versus using a coarse lock. For small numbers of CPUs, the performance grows slightly as CPUs are added since the entire pool does not fit into a single CPU's cache, and some cache miss penalty can be hidden under the lock-waiting. But as expected, coarse grain locking leads to very poor throughput when the number of CPUs grows further (note the step-functions as the number of CPUs crosses the chip and MCM boundaries). In contrast, transactions scale very well. Even at 100 CPUs, the performance is not limited by the concurrency, but by the cache miss penalty that almost every iteration incurs when accessing a cache line that was previously accessed by another CPU: at 100 CPUs, the throughput with TBEGINC is 99.8% of the throughput without any locking scheme.

While not particularly interesting for real-world commercial applications, we also studied the performance of transactions versus

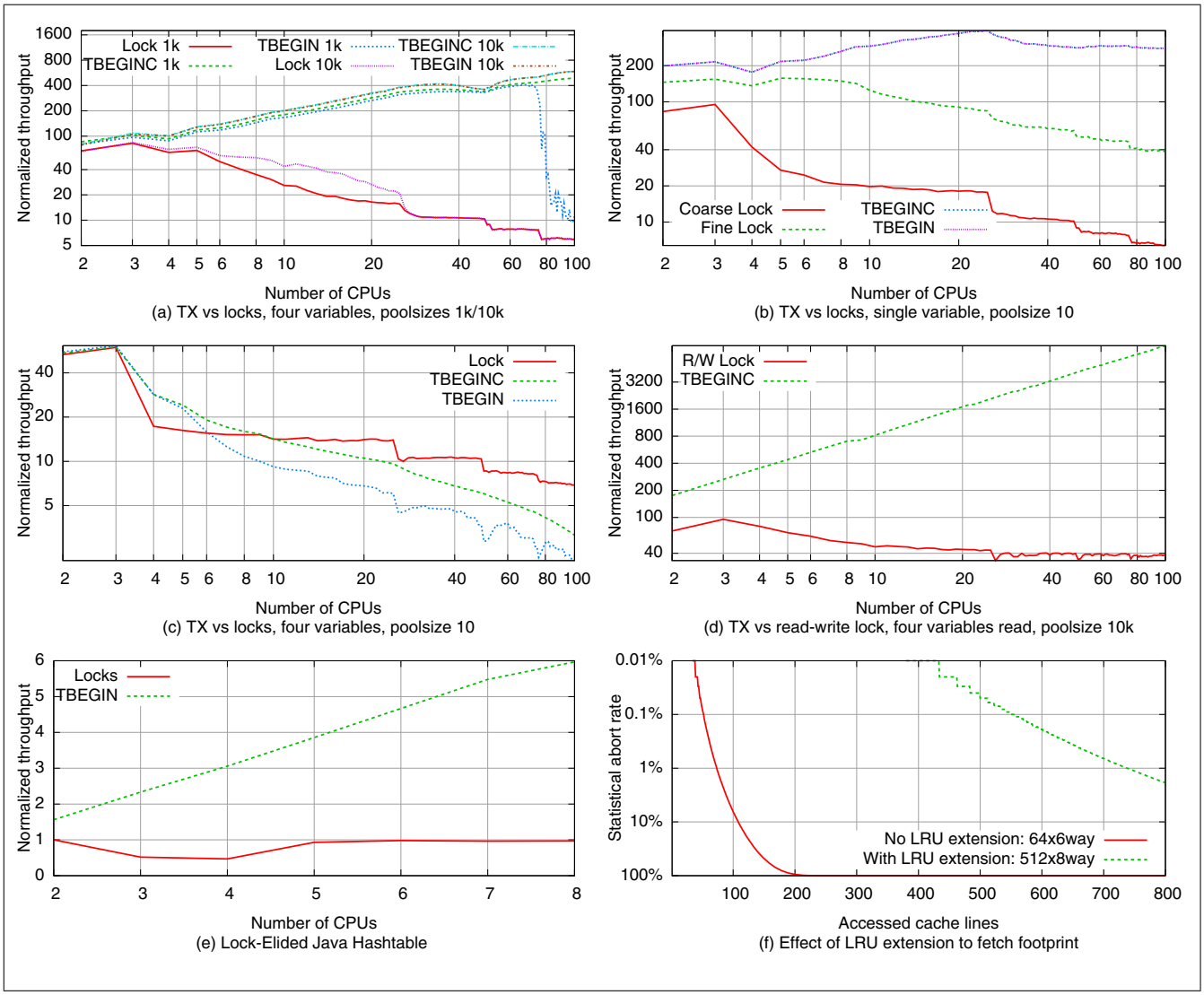

Figure 5: Performance Results

locks for very high contention cases. As can be seen in figure 5(a), when using a pool of 1000 lines, the throughput using TBEGIN drops steeply after the number of CPUs reaches a threshold, but still exceeds the locking performance. Figure 5(b) shows the performance of updating a single variable from a very small pool of only 10 variables. As expected, coarse grain locks yield very poor throughput. The throughput is better with fine grained locks, although it does not grow much with the number of CPUs and declines with more than 10 CPUs. In contrast, with transactions, the throughput grows up to 24 CPUs (the size of the MCM node in the tested system) and holds steady beyond, and transactions out-perform locks across the entire CPU range. Figure 5(c) shows updating of 4 variables from a pool of 10. With up to 6 CPUs, transactions behave slightly better than a coarse grained lock. But as the number of CPUs and such the contention grows further, locks perform better, not dropping as steeply as transactions. The reason for the difference between single-variable and four-variable updates is that for the latter case a CPU must receive all 4 lines into the L1 cache before it can commit the transaction; after the first line is received, the increment instruction for that variable can execute out-of-order. The transaction then becomes subject to conflicts on that cache line while the CPU is waiting for the other cache lines. This leads to a high abort rate, which means that many cache line transfers in the system are wasted. In contrast, as soon as a CPU obtains a lock, that CPU is guaranteed to finish the update on all 4 lines and thus less cache bandwidth is wasted.

It is interesting to note that under extreme contention, constrained transactions behave better than non-constrained transactions. This is because the CPU turns off speculative fetching after a certain number of aborts for constrained transactions, preserving some cache bandwidth which helps throughput. We did not implement this feature for performance reasons, but in order to guarantee eventual success for constrained transactions.

The above test cases all update shared variables. Another impor-

tant case is reading of shared variables. Traditionally a read-write lock is used if updates are relatively rare. This allows shared access to the variables by multiple CPUs while no updates are in progress. Typical implementations of read-write locks require updating of the lock-word every time a reader enters or leaves its critical section, in order to keep track of how many readers are in-flight. The update of the read-count causes the lock-word to be transferred between CPUs, which limits the throughput significantly (see figure 5(d)). Transactions avoid this problem since they only need to check the write-count to be 0, without updating the read-count. If a writer enters during the transaction, the writer's update of the write-count causes all reader transactions to abort, but as long as no writers appear, all CPUs can share the read/write count cache line. This leads to almost linear performance improvement with the number of CPUs.

Figure 5(e) shows the performance of a more real-world example. The IBM Java team has prototyped an optimization in the IBM Testarossa JIT to automatically elide locks used for Java synchronized sections. This optimization has been shown to transparently improve the scalability of widely used standard data structures such as `java/util/hashtable`. Multiple software threads run under z/OS, accessing the hash table for reading and writing. As can be seen in figure 5(e), the performance using locks is flat, whereas the performance grows almost linearly with the number of threads using transactions.

In another experiment (not shown in figure 5), the Java team has implemented the `ConcurrentLinkedQueue` using constrained transactions. The throughput using transactions exceeds locks by a factor of 2. In [23], the IBM XL C/C++ team compares a subset of the STAMP benchmarks using pthread locks and transactions. Depending on the benchmark application, transactional execution improves performance by factors between 1.2 and 7.

As described in section 3.3, the L1 cache employs a LRU-extension scheme to enhance the supported fetch footprint beyond the L1 cache size. Figure 5(f) shows the statistical abort rate (%) from associativity conflicts with n=1...800 accesses to random congruence classes. As can be seen, the abort rate for large transactions is significantly reduced when the footprint limitation is moved from the L1 cache to the L2 cache, as is done by the L1 cache LRU extension scheme. Of course, very large and long transactions may suffer from other abort reasons like conflicts with other CPUs, LRU evictions from higher level caches, or asynchronous interrupts. These effects limit the practical transaction size. Learning over time will show exactly how to best tailor transactions, but we feel that with the LRU extension, the read footprint will not be a limitation.

V. Summary

We have described the instruction-set architecture and implementation of the transactional memory feature of the latest mainframe server processor in the IBM zEC12 system. Special focus was put on software test and debug, as well as the introduction of transactional memory support into an existing SMP and microprocessor design. The transactional memory feature is defined so that integration into existing large-scale software products can be done without a complete software redesign. For example we have shown how transactions and locks can co-exist by eliding locks using transactions. The introduction of constrained transactions eases the explitation of transactional memory be removing the need for a lock-based fallback path.

We have evaluated the performance on a set of micro-benchmarks, and under realistic contention, the performance of the transactional memory system meets our expectations and clearly exceeds the performance of traditional lock based methods, in some cases significantly. Preliminary performance experiments with examples like a parallel hash table are very promising and show almost linear n-way scalability.

The IBM compiler development teams are involved in ongoing development for support of transactional memory in various programming languages. In [24], the support of transactional memory in IBM's XL C/C++ compiler is described. The operating systems and middleware development groups are aggressively identifying opportunities to improve the scaling of hot-spots. Transactional memory is a very promising new tool for improving parallel software scalability, that is driving innovation in both hardware and software design now and in the future.

References

[1] http://www-03.ibm.com/press/us/en/pressrelease/38653.wss

[2] K. Shum, IBM zNext – The 3rd Generation High Frequency Microprocessor Chip, HotChips, 2012

[3] M. Herlihy, J. Eliot, B. Moss, Transactional Memory: Architectural Support for Lock-Free Data Structures, ISCA, 1993

[4] L. Hammond et al., Transactional Memory Coherence Consistency, ISCA, 2004

[5] K. Moore, J. Bobba, M. Moravan, M. Hill and D. Wood, LogTM Log-Based Transactional Memory, HPCA, 2006

[6] S. Chaudhry et al., Rock: A high-performance Sparc CMT processor. IEEE Micro, 29(2):6-16, 2009

[7] R. Haring et al., The IBM Blue Gene/Q Compute Chip. IEEE Micro 32(2): 48-60 (2012)

[8] http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/

[9] A. McDonald et al., Architectural Semantics for Practical Transactional Memory, ISCA, 2006

[10] J. Chung et al., Tradeoffs in Transactional Memory Virtualization, ASPLOS, 2006

[11] T. Harris et al., Transactional Memory: An Overview, IEEE Micro 27:3, May 2007

[12] R. Rajwar, J. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. MICRO, 2001

[13] M. Herlihy. A methodology for implementing highly concurrent data objects. In ACM Transactions on Programming Languages and Systems 15:5, 1993

[14] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, C. Zilles. Hardware Atomicity for Reliable Software Speculation. ISCA, 2007

[15] Y. Lev, M. Moir. Debugging with transactional memory. TRANSACT '06: 1st Workshop on Languages, Compilers, and Hardware Support for Transactional Computing, 2006.

[16] z/Architecture Principles of Operation, IBM Publication SA22-7832-09, 2012

[17] R. Guerraoui, M. Kapalka. On the correctness of transactional memory. Symposium on Principles and Practice of Parallel Programming (PpoPP), 2008

[18] D. Dice, O. Shalev, N. Shavit, Transactional locking II, DISC 06: 20th International Symposium on Distributed Computing, 2006

[19] C. Blundell, E. C. Lewis, and M. Martin. Deconstructing transactions: The subtleties of atomicity. WDDD 05: 4th Annual Workshop on Duplicating, Deconstructing, and Debunking, 2005

[20] F. Busaba, M. Blake, B. Curran, M. Fee, C. Jacobi, P.-K. Mak, B. Prasky, C. Walters, IBM zEnterprise 196 microprocessor and cache subsystem, IBM Journal of Research and Development, Vol 56:1.2, 2012

[21] P. Mak, C. Walters, G. Strait, IBM System z10 processor cache subsystem microarchitecture, IBM Journal of Research and Development, Vol 53:1, 2009

[22] K. Choy, J. Navarro, C.-L. Shum, A. Tsai, Method, System, and Computer Program Product for Cross-invalidation Handling in a Multi-level Private Cache, US Patent Application 20090240889

[23] M. Mitran, V. Vokhshori, Evaluating the zEC12 Transactional Execution Facility, IBM Systems Magazine, 2012

[24] M. Mitran, V. Vokhshori, IBM XL C/C++ Maximizes zEC12's Transactional Execution Capabilities, IBM Systems Magazine, 2012