Transactional Memory

Contents of Lecture 9

- Transactional Memory: HTM and STM
- HTM in Blue Gene/Q, POWER8, EC12 and Intel Core 7-4770
- STM with Clojure

TM: Transactional Memory

- A transaction is a sequence of reads and writes which either occur atomically or not at all
- Consider the swish example:
  - With many more accounts than threads there is little risk of data races
  - Little risk is different from impossible
- The idea with transactional memory is to take a chance without locks
- If there are conflicting accesses then try again
- What programmers need to use transactional memory is to identify:
  - the start of a transcation
  - the end of a transcation
- Hardware or software implementation of detecting conflicts and managing transactions

Using Transactional Memory

- The programmer "only" has to divide a program into transactions.
- No need for programmer orchestration using locks etc.
- Performance tuning is based on feedback and results in changing the transactions — which will affect only performance and not correctness.
- With too many conflicts and restarted transactions, the program will be very slow
- In C with `gcc -fgnu-tm swish.c`

```
__transaction_atomic {
        from->balance -= amount;
        to->balance += amount;
}
```

C++

- *ISO/IEC TS 19841:2015 describes extensions to the C++ Programming Language that enable the specification of Transactional Memory.*
- It is a "non-normative extension" to C++ and may be included in ISO C++ in the future
- See: `https://www.iso.org/standard/66343.html`

Software TM, STM

- Software defines the transactions (e.g. one loop iteration).
- Transactions log accesses to shared data (each operation)
- Before committing a transaction, it must check that no violations happened
- The main problems of STM are maintaining the logs and doing the commit
- Do all memory accesses have to be logged?

Reducing the amount of logging

- The programmer can annotate reads and writes as being to private (or shared) data
- The compiler can perform analysis to avoid some logging
- The language may permit STM only for certain data type
- For C/C++ STM is likely to be tough without programmer annotations:
  - Stack accesses are trivial for the compiler
  - But which lists or trees are accessed by only one thread/transaction?
- In Clojure, a certain kind of pointer, a `ref`, can only be modified in a transaction

Experience from IBM Research at Austin in 2008

- After spending two years implementing STM they wrote an article with the title

  Software Transactional Memory: why is it only a research toy?

  - Each shared access expands to tens of additional machine instructions
  - Reusing memory allocated and freed by malloc/free cannot always be done since the STM system is not informed
  - Transactions using legacy code such as third-part libraries can require serializing transactions
  - Debugging is complicated due to non-determinism

Hardware TM

- Again, software defines the transactions (e.g. one loop iteration).
- Hardware somehow buffers all writes locally
- When a transaction has completed, hardware commits it
- At commit, all local writes are transferred to the shared memory
- Other processors listen to the commit and can detect that its transaction has violated a dependency and needs to be restarted.
- The conflict detection granularity typically is a cache block so false sharing can create conflicts due to false sharing!

Performance Programming for TM

- Minimize violations: don't write transactions that access shared variables too much for too long.
- On the other hand, making transactions too small introduces overhead.
- Avoid buffer overflows. If a processor's own buffer overflows, the transactions must "swap" the local writes to memory (of course not making the data visible to others — just in order to get more storage).

Privatization in TM

- All shared data must be accessed only in transactions.
- It's a data-race to let both non-transactions and transactions access the same variable.
- Sometimes, however, there is shared data that was accessed in transactions but after a certain point it will only be accessed by one thread.
- Such data should be **privatized** in order to:
  - Avoid overhead of accesses in transactions.
  - Enable non-reversible actions in transactions — e.g. I/O

Transactional Memory vs locking

- No deadlocks for TM but poor performance if there are many conflicts.
- Data must be partitioned for good performance:
  - avoiding lock contention,
  - avoiding transaction conflicts
- Critical section performance
  - full speed with locking — contention at synchronization only
  - contention can degrade performance anywhere during transaction
- Debugging:
  - natural with locks
  - it can be more difficult to set break points in the middle of a transaction — depending on the hardware support
- Privatization:
  - trivial with locks
  - TM needs hardware support or performance penalty

Tom Knight: inventor of transactional memory

- Studied partly at MIT from age 14
- Built network interface to the 6th computer connected to ARPANET
- Registered first .com address: `symbolics.com`
- Worked on Ethernet and Lisp at MIT
- Worked for Thinking Machines
- Founder of the scientific field synthetic biology
- Published a paper 1986 about transactional memory for Lisp
- So: suitable that we use Clojure for transactional memory
- 1986 was also the year when Michel Dubois published the paper on Weak ordering

Sun's Rock processor

- Sun was founded 1982 and pioneered the workstation market (and later created Java)
- Rock was the first implementation of transactional memory
- Presented at the main computer architecture conference, ISCA, in 2009
- One of the authors, Anders Landin, was a D-student at LTH (and then went to SICS to participate in research on cache-only memory architectures, COMA, in the DDM project)
- In the 2008 financial crises many of Sun's customers went out of business
- Oracle bought Sun
- It was produced as prototypes but canceled by Larry Ellison

The IBM Blue Gene/P supercomputer

The IBM Blue Gene/P supercomputer

- USD 1.3 million per rack
- 16 cores and SMT-4, clocked at 1.6 GHz
- The IBM Blue Gene/P supercomputer was the first commercial machine that implemented transactional memory in hardware.
- Recall that superscalar processors don't allow speculative instructions to modify memory.
- In Blue Gene/P they are allowed to write to the cache by also writing a version tag.
- With the version tag, aborted transactions can be rolled back.

The IBM Blue Gene/P basic design

- Each chip has 16 cores with four hardware threads
- Each core has a 16 KB L1 cache
- All cores share a 32 MB L2 cache
- Speculative writes are saved in the L2 cache
- They become visible after a successful commit

Conflict detection in the L2 cache

- Conflicts are detected the cache coherence protocol and L2 caches
- A thread stores its version of a cache block in the L2 cache
- Accesses are marked as reads or writes, and speculative or not
- For speculative accesses it is in addition marked which other threads have accessed a block
- Two modes: either short or long transactions — but not both
- The normal granularity is 64 bytes
- Cache block and cache line are the same

Short transactions

- As we will see below, transactions can be non-speculative
- This slide applies to speculative transactions
- At an L1 cache load, the L2 cache is informed (otherwise it cannot detect conflicts)
- At an L1 cache store, that L1 cache block is removed from the L1 cache, and in this case moved to the L2 cache since it was just modified
- After a store, a subsequent load gets an L1 cache misse
- This is expected to work OK for short transactions which do not reuse speculative data very much

Long transactions

- Long running transactions can use the L1 cache for speculative state
- In order to make the L2 cache know about an initial access to such shared state, the entire L1 cache is invalidated at the start of a transaction
- Note that multiple hardware threads can save different versions of some data in the L1 cache
- This is done by using a trick with the TLB: different virtual to physical address translations are used for different transactions so the data looks different to the L1 cache
- TLB = translation lookaside buffer invented for IBM 360 in the 1960s
- For long running transactions which reuse data, invalidating the L1 cache at start is expected to be OK

Runtime support

- Transactions must be single-entry and single-exit
- No exceptions in transactions are allowed
- The stack pointer and three other registers must always be saved and passed to the kernel since it needs them if a transaction fails
  - a pointer used for the GOT (global offset table) table for position independent code
  - the instruction address for the register restore code
  - a copy of the time base register (see `timebase.c` in Tresorit which is used for very accurate timing)
- Thus, a system call (expensive thing) is involved.
- Which other registers need to be saved/restored are determined by the compiler

Abort and retry

- If there is a conflict the kernel is invoked
- The kernel uses the saved time base register value to determine which of two transactions is the oldest
- The age is used as a priority to decide which transaction should abort
- The older transaction is normally selected to survive
  - it has probably done more work already
- The cancelled transaction is retried

Irrevocable transactions

- After too many retries the runtime system switches transaction mode to irrevocable
- Irrevocable transactions cannot fail (unless they crash of course)
- An irrevocable transaction starts with taking the **irrevocable token** which is a global lock
- A completed irrevocable transaction may at runtime be marked as **problematic** (meaning the instruction address of the transaction).
- Problematic transactions switch to irrevocable after first failure

Evaluation

- IBM used the STAMP benchmarks from Stanford
- See `https://github.com/kozyraki/stamp`
- STAMP is available with transactions, OpenMP directives, and as single-threaded C/C++ codes
- IBM compared the Blue Gene HTM with:
  - single thread execution
  - manually optimized OpenMP
  - manually marked numerous reads and writes as non-shared for STM
- When evaluating something, the numbers are of practical interest (is it worth buying/producing/etc something?) but it is the explanations to why we see the numbers that matters — this is equally for important for research papers as for MSc theses
- Don't write: "we saw X was Y % better for Z benchmarks but we did not have time to figure out why" — its translation is "we could not figure out why" :)

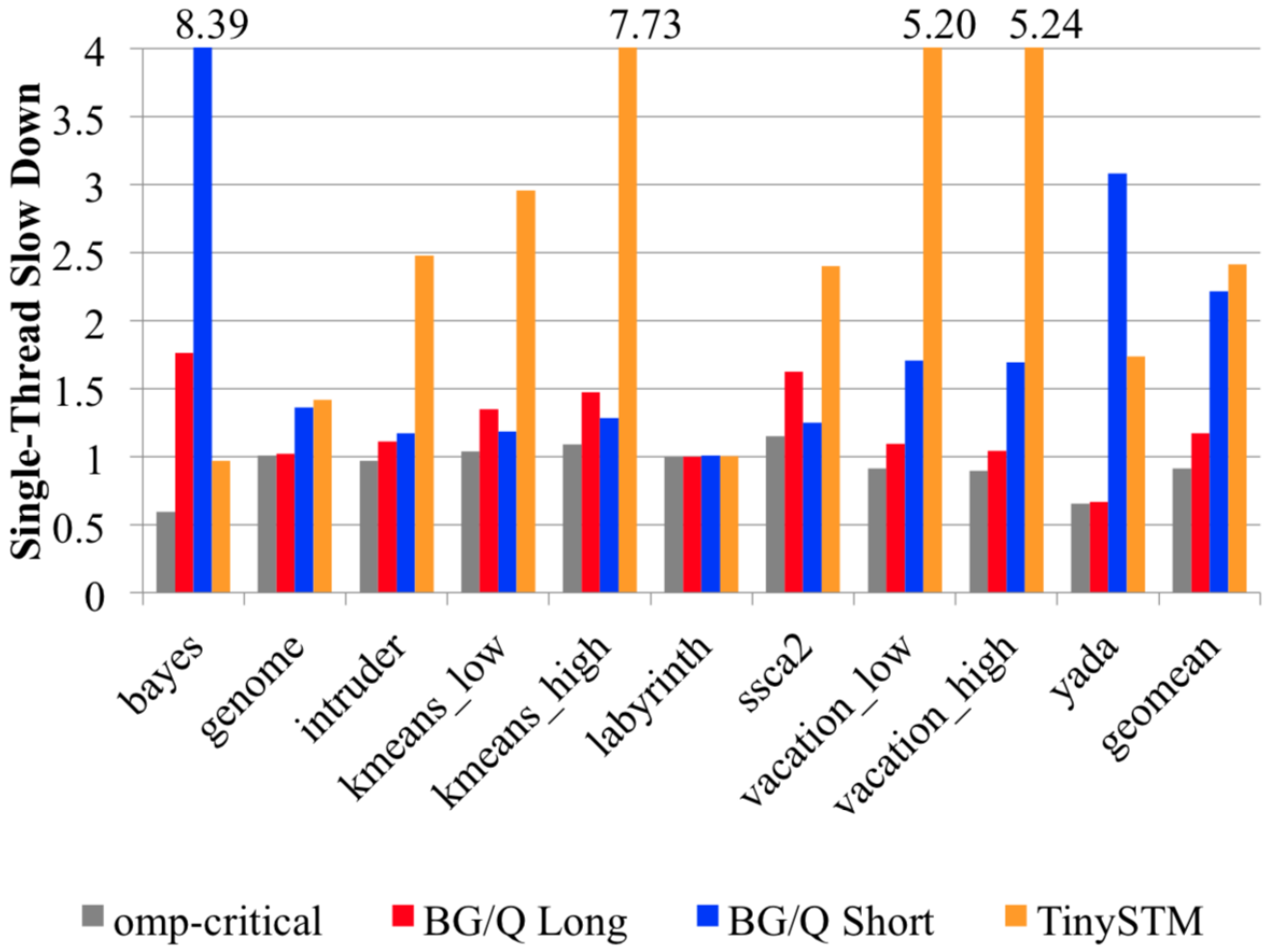
L1 cache misses

Benchmark	# L1 misses per 100 instr. (thread=1)			Instr. path length relative to serial (thread=1)		
	sequential	BG/Q Short	BG/Q Long	omp critical	BG/Q Short	BGQ Long
bayes	0.6	8.1	0.7	87 %	155 %	156 %
genome	0.6	1.6	0.7	99 %	101 %	101 %
intruder	0.8	1.5	1.3	97 %	102 %	104 %
kmeans_low	0.1	0.3	2.7	101 %	105 %	106 %
kmeans_high	0.1	0.7	3.2	103 %	110 %	113 %
labyrinth	0.9	1.0	1.2	100 %	100 %	101 %
ssca2	1.0	0.7	1.3	96 %	109 %	111 %
vacation_low	1.5	6.2	2.4	88 %	92 %	93 %
vacation_high	1.7	7.0	2.4	88 %	91 %	92 %
yada	1.5	7.2	0.8	101 %	90 %	90 %

Table 2: Hardware performance monitor stats for the STAMP benchmarks.

- Many more L1 cache misses in TM code
- Instruction path lengths = number of executed instructions

Single thread slowdown



- L1 cache misses penalize BG /Q Short
- OpenMP sometimes better due to better register allocation, i.e. luck

Speedups: sometimes good but none very efficient

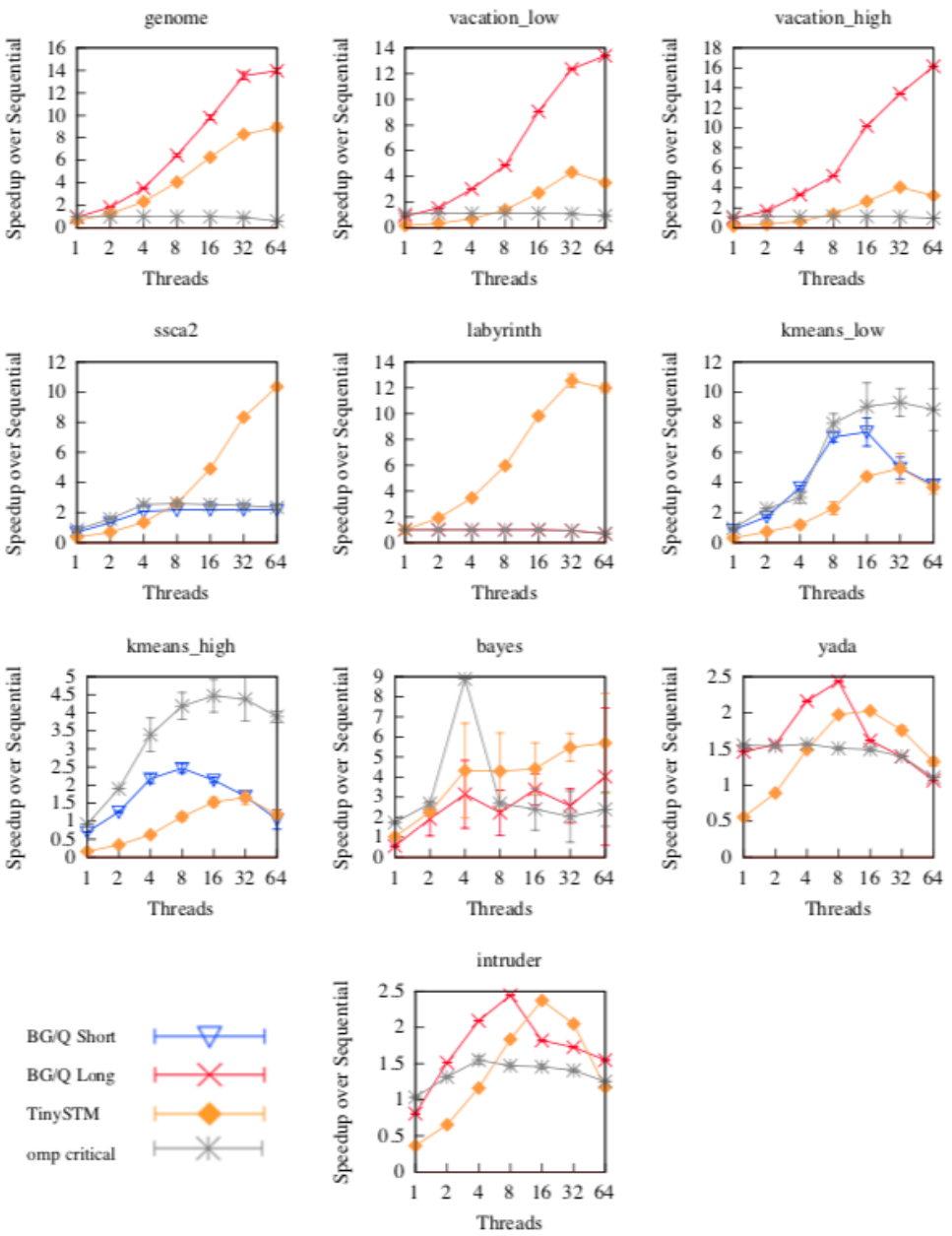


Figure 5: Speedup over sequential for upto 64 threads.

Power 2.07 supports Transactional Memory

- The Power architecture has supported TM since version 2.07, published May 10, 2013.
- Our POWER8 supports this
- Every memory access is either transactional or non-transactional.
- New instructions include (the `.` suffix means they set the condition codes in `CR0`):
  - `tbegin.`
  - `tend.`
  - `tabort.`
  - `tsuspend.` leave the transaction
  - `tresume.` return to the transaction
  - `treclaim.` used by kernel at context switches
  - `trechkpt.` used by kernel to copy certain registers
- Memory accesses executed between the `tbegin.` and `tend.` are transactional and all other are non-transactional.

Overview

- The bit `TDOOMED` is set to 0 by `tbegin.` and to 1 at a failure.
- Most registers (but not `CR0` which returns success/fail) are saved at a `tbegin.` and are restored at a transaction failure.
- A failure handler is run at a transaction failure.
- A transaction can fail either due to itself or due to another transaction.
- It fails by itself (called self-induced) if:
  - It executes `tabort.`
  - It has a too deep transaction nesting level at a new `tbegin.`.
  - It has a too large footprint (written too much data).
  - It executes a disallowed instruction — such as `doze`, `sleep` and `dcbi`.
- The failure handler should either retry the transaction or do the operation without a transaction (i.e. with locks).
- There is no guarantee of any progress or fairness by the hardware.

Nested transactions

- Transactions can be nested.
- A `tend.` with field `A=1` ends all transactions of the thread and with `A=0` only ends the most recently started.
- A failure of a nested transaction terminates all transactions!

Conflicts

- A transaction conflicts with another transaction or a non-transactional access if they access the same cache block (i.e. memory block) and at least one is a store.
- At least one of two conflicting transactions fail, i.e. are aborted.
- Note the cache block granularity: since the cache block size is not defined by the architecture, software must be written accordingly.
- The reason for transaction failure is provided in a register.

Transaction execution states

- There are three states:
  - Non-transactional: the normal state before any transaction is started.
  - Transactional: execution between a `tbegin.` and a `tend.`.
  - Suspended: execution by the same thread but as a temporary escape of the transactional state. This is execution between a `tsuspend.` and a `tresume.`
- The purpose of the suspended state is for instance
  - Inter-thread communication: cannot be rolled back!
  - Other stores which should not be rolled back such as for debugging.
  - Accesses to non-cachable memory.
- Code in suspended state should be careful when accessing transactionally modified data.

Suspend confusions

```
x = 0;
tbegin
x = 1;
tsuspend
x += 1
tresume
tend
```

- The += stores 2 and kills the transaction so x goes from 0 to 2.
- This is non-intuitive since it x "never" was 1
- IBM manual about using the processor suspended state:
  - *Accessing storage locations in Suspended state that have been accessed transactionally has the potential to create apparant storage paradoxes.*
  - *It must be used with care.*

POWER8 TM

- POWER8 implements the POWER 2.07 architecture specification
- As in BG/Q the L2 cache is used for speculative state
- The L1 data cache is 64 KB and the L1 instruction cache is 32 KB
- The cache block size is 128 bytes in all caches
- All caches are 8-way associative
- All up to 8 hardware threads in a core share the same L1 and L2 caches
- Each core has its own L2 512 KB cache
- The L3 cache is 8 MB and there is one per chip, i.e. one in power.cs.lth.se

Other HTM implementations

- Intel's Transactional Synchronization Extensions (TSX) were used in some Haswell processors in 2013
- In 2014 a bug was detected and TSX was disabled with a microcode update on these chips
- TSX is a software API and the hardware details are not public
- Some Skylake processors support it
- Both AMD and ARM also have HTM
- IBM mainframes (updated but in production since 1952!) also implement HTM, such as EC12
- EC12 has 6 cores clocked at 5.5 GHz

Comparing BG/Q, POWER8, EC12 and Intel Core 7-4770

- In 2015 IBM Research in Tokyo and Austin compared these on STAMP
- Some "TM unfriendly" parts of STAMP were fixed
- The number of used retries were tuned to each machine
- All machines detect conflicts using the cache coherence protocol

Differences

- On BG/Q programmers cannot write code to handle failed transactions
- On BG/Q system calls are used to begin and commit transactions
- The other machines use normal machine instructions
- EC12 has the largest cache lines, 256 bytes, and highest risk for false conflicts
- EC12 uses also the L1 cache to detect conflicts (i.e. in addition to the L2 cache)
- Intel uses also the L1 cache to detect conflicts in addition to other resources
- When the transaction capacity is exceeded, the transaction is aborted
- The transaction capacity of Intel is not public but experiments revealed it to be 4 MB for reads and 22 KB for writes
- POWER8 has a combined transaction capacity of 8 KB only

Differences

Table 1. HTM implementations of Blue Gene/Q, zEC12, Intel Core i7-4770, and POWER8

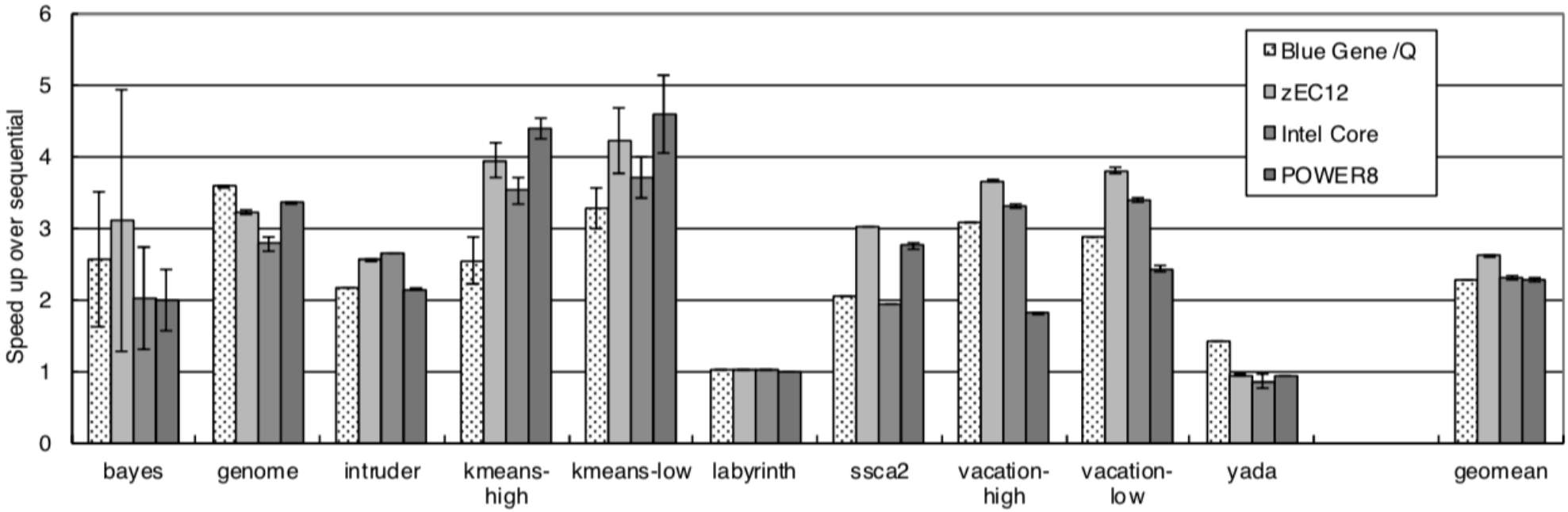
Processor type	Blue Gene/Q	zEC12	Intel Core i7-4770	POWER8
Conflict-detection guranularity	8 - 128 bytes	256 bytes	64 bytes	128 bytes
Transactional-load capacity	20 MB (1.25 MB per core)	1 MB	4 MB	8 KB
Transactional-store capacity	20 MB (1.25 MB per core)	8 KB	22 KB	8 KB
L1 data cache	16 KB, 8-way	96 KB, 6-way	32 KB, 8-way	64 KB
L2 data cache	32 MB, 16-way, (shared by 16 cores)	1 MB, 8-way	256 KB	512 KB, 8-way
SMT level	4	None	2	8
Kinds of abort reasons	-	14	6	11

Evaluation

CPU	cores	SMT	clock
Intel Core 7-4770	4	2	3.4 GHz
IBM EC12	16	1	5.5 GHz
IBM BG/Q	16	4	1.6 GHz
IBM POWER8	6	8	4.1 GHz

- The POWER8 is available with 6 to 12 cores
- To make comparisons fair, only four cores where used in any machine
- The performance metric is speedup over single thread for the same machine
- Thus no comparison in execution of IBM vs Intel since the machines are so different
- The purpose is to understand what to improve in future TM implementations

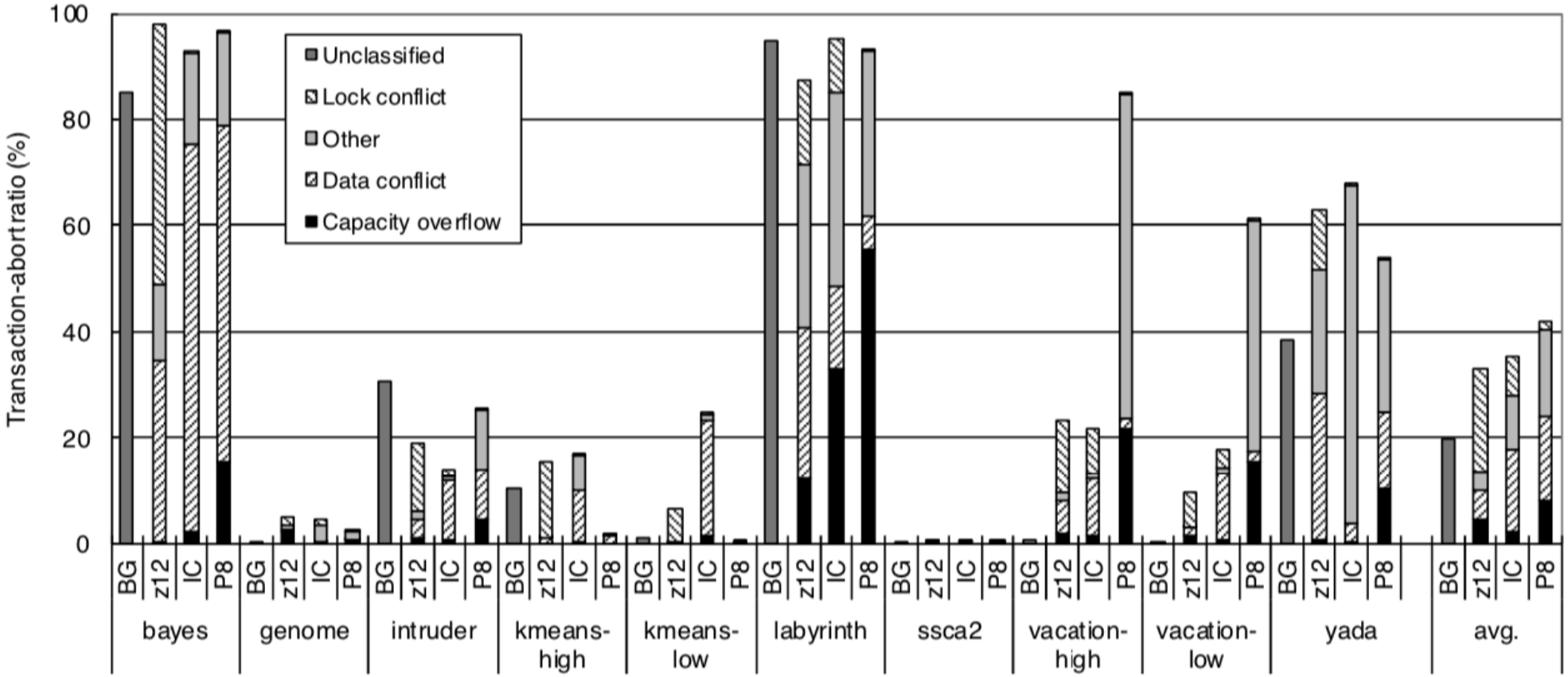
4 core speedups vs itself



Transaction failures — remarks

- Intel performs hardware prefetching which are counted as transaction accesses
- This causes a higher number of data conflicts than the IBM machines
- This is significant for kmeans-low
- The IBM researchers disabled hardware prefetch on Intel to verify this
- Then informed Intel which confirmed the findings
- POWER8 had more capacity failures so the capacity should be increased
- *The zEC12 suffers from mysterious transaction aborts that degrade its performance*
- The failure codes where not documented and happened in odd situations
- This is a better explanation than saying "did not do it due to lack of time"!

Transaction failures



Clojure

- Clojure is a Lisp language with support for software transactional memory
- To use STM you need to use a special type, called `ref`
- This has the advantage that only such objects need to be logged

```
(def start-balance 1000)
(defrecord account [balance])
(def pointer (ref (->account start-balance)))

(println (deref pointer))
(println @pointer)          ; @ means deref

(update @pointer :balance + 5)

(println @pointer)          ; prints 1000
```

- This creates a new object with balance 1005
- But does not modify `pointer`

ref-set

- `ref-set` is used to modify a `ref`

```
(def start-balance 1000)
(defrecord account [balance])
(def pointer (ref (->account start-balance)))

(println (deref pointer))
(println @pointer)          ; @ means deref

(ref-set pointer (update @pointer :balance + 5))

(println @pointer)
```

- This does not work
- We can only modify a ref in a transaction

dosync

- The following works

```
(def start-balance 1000)
(defrecord account [balance])
(def pointer (ref (->account start-balance)))

(println @pointer)

(dosync (ref-set pointer (update @pointer :balance + 5))
        (ref-set pointer (update @pointer :balance + 6))
        (ref-set pointer (update @pointer :balance + 7)))

(println @pointer)
```

- The balance becomes 1018