Contents of Lecture 10

- More about compiler-based automatic parallelization
- Hardware-based automatic parallelization: research at Stanford
- Transactional Memory: HTM and STM
- HTM in Blue Gene/Q, POWER8, EC12 and Intel Core 7-4770
- STM with Clojure
The set of all vectors of dependence distances is represented by the distance matrix $D$.

We are free to swap the rows of $D$ since it really is a set of dependences.

Unimodular transformations require that all dependences are uniform, i.e. with known constants.

Consider a uniform dependence vector $d = j - i$.

With index variables $K = I U$ we have $d_U = jU - iU = dU$.

Therefore, given a dependence matrix $D$ and a unimodular transformation $U$, the dependences in the new loop $L_U$ become:

$$D_U = DU$$
Valid Distance Matrices

- The sign, **lexicographically**, of a vector is the sign of the first nonzero element.
- A distance vector can never be lexicographically negative since it would mean that some iteration would depend on a future iteration.
- Therefore no row in the new distance matrix $D_U = DU$ may be lexicographically negative.
- If we would discover a lexicographically negative row in $D_U$, that loop transformation is invalid, such as the second row of the following $D_U$:

$$D_U = \begin{pmatrix} 1 & 2 \\ -1 & 1 \end{pmatrix}$$
By **outer loops** is meant all loops starting with the outermost loop.

While we always can find a unimodular matrix through which we can parallelize the inner loops, this is not the case for outer loops.

To parallelize the inner loops, we need to assure that all loop carried dependences are carried at the outermost loop.

In other words, the leftmost column of the distance matrix $D_U$ simply should consist only of positive numbers!

For outer loop parallelization, $D_U$ instead should have leading zero columns.
A column of a matrix is linearly independent if it cannot be expressed as a linear combination of the other columns.

The rank of a matrix is the number of linearly independent columns.

For instance, an identity matrix $I_m$ with $m$ columns has rank($I_m$) = $m$.

Any unimodular $m \times m$-matrix $U$ has rank($U$) = $m$.

A matrix with zero columns must have a rank less than the number of columns.

So, since $D_U = DU$, if $D_U$ should have a rank less than $m$, it must be $D$ which contributes with that.
Outer Loop Parallelization Example

- Assume we have the distance matrix $D$ defined as:

$$D = \begin{pmatrix} 6 & 4 & 2 \\ 0 & 1 & -1 \\ 1 & 0 & 1 \end{pmatrix}$$

- With this distance matrix, only the innermost loop can be executed in parallel.

- E.g. the above row above $(0, 1, -1)$, is lexicographically positive

- We want a $D_U$ with lexicographically positive rows and at least one zero column to the left.

- For example:

$$D_U = \begin{pmatrix} 0 & ? & ? \\ 0 & ? & ? \\ 0 & ? & ? \end{pmatrix} = \begin{pmatrix} 6 & 4 & 2 \\ 0 & 1 & -1 \\ 1 & 0 & 1 \end{pmatrix} U$$

- If $\text{rank}(D) = 3$ then such a $U$ cannot exist.
Steps towards Finding $U$

- We start with transposing $D$:
  $$D^t = \begin{pmatrix}
6 & 0 & 1 \\
4 & 1 & 0 \\
2 & -1 & 1
\end{pmatrix}$$

- Using the Echelon reduction algorithm, we compute:
  - a unimodular matrix $V$
  - an echelon matrix $S$

- Such that $VD^t = S$, e.g.
  $$\begin{pmatrix}
0 & 0 & 1 \\
0 & 1 & -2 \\
1 & -1 & -1
\end{pmatrix}D^t = \begin{pmatrix}
2 & -1 & 1 \\
0 & 3 & -2 \\
0 & 0 & 0
\end{pmatrix}$$
More Steps towards Finding $\mathbf{U}$

- We have $\mathbf{VD}^t = \mathbf{S}$:
  $$
  \begin{pmatrix}
  0 & 0 & 1 \\
  0 & 1 & -2 \\
  1 & -1 & -1 \\
  \end{pmatrix}
  \begin{pmatrix}
  6 & 0 & 1 \\
  4 & 1 & 0 \\
  2 & -1 & 1 \\
  \end{pmatrix}
  =
  \begin{pmatrix}
  2 & -1 & 1 \\
  0 & 3 & -2 \\
  0 & 0 & 0 \\
  \end{pmatrix}
  $$

- Assume we wish to find $n = 1$ parallel outer loops.

- Then we find an $m \times (n + 1)$ matrix $\mathbf{A}$ such that $\mathbf{DA}$ has $n$ zero columns and then a column with elements greater than zero.

- This $\mathbf{A}$ will be used to find $\mathbf{U}$.

- How can we find $\mathbf{A}$?

- Multiplying the last row of $\mathbf{V}$ with the columns of $\mathbf{D}^t$ produces the zero row in $\mathbf{S}$.

- Thus, the first column of $\mathbf{A}$ should be the last row of $\mathbf{V}$, i.e.
  $$
  \mathbf{DA} = 
  \begin{pmatrix}
  6 & 4 & 2 \\
  0 & 1 & -1 \\
  1 & 0 & 1 \\
  \end{pmatrix}
  \begin{pmatrix}
  1 & \ ? \\
  -1 & \ ? \\
  -1 & \ ? \\
  \end{pmatrix}
  =
  \begin{pmatrix}
  0 & \ ? \\
  0 & \ ? \\
  0 & \ ? \\
  \end{pmatrix}
  $$
Finding the last column of $A$ is easy. Denote it $u$.

$$DA = \begin{pmatrix} 6 & 4 & 2 \\ 0 & 1 & -1 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & u_1 \\ -1 & u_2 \\ -1 & u_3 \end{pmatrix} = \begin{pmatrix} 0 & \geq 1 \\ 0 & \geq 1 \\ 0 & \geq 1 \end{pmatrix}$$

Multiplying each row of $D$ with $u$ should produce a positive number:

$$6u_1 + 4u_2 + 2u_3 \geq 1$$

$$u_2 - u_3 \geq 1$$

$$u_1 + u_3 \geq 1$$

We find $u$ to be e.g. $u = (1, 1, 0)$.

$$A = \begin{pmatrix} 0 & 1 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$$
Given a matrix $A$, using a variant of the algorithm for echelon reduction, we can find a unimodular matrix $U$ such that $A = UT$

i.e. 

$$A = \begin{pmatrix} 0 & 1 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} = UT = \begin{pmatrix} -1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} -1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$$
Computing $L_U$

- With this loop transformation matrix $U$, we get the following new dependence matrix $D_U$:
  
  $$D_U = DU$$

- i.e.
  
  $$D_U = \begin{pmatrix} 0 & 10 & 6 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} = DU = \begin{pmatrix} 6 & 4 & 2 \\ 0 & 1 & -1 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} -1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

- The compiler does not actually need to compute $D_U$ but it is a nice internal check to verify no row is lexicographically negative.

- The new loop $L_U$ is constructed as explained before:

- A loop nest $L$ is changed to a new loop nest $L_U$ with loop index variables:

  $$K = IU$$

- New array references and new loop bounds must be computed.

- We have already seen both of these two, but repeat them for convenience on the next two slides.
Recall: Computing the New Index Variables

- With

\[ p_0 \leq IP \qquad IQ \leq q_0 \]

\[ I = KU^{-1} \]

We use Fourier-Motzkin elimination to find the loop bounds from

\[ p_0 \leq KU^{-1}P \qquad KU^{-1}Q \leq q_0 \]

- The bounds are found starting with \( k_1, k_2 \) etc.
All array references are rewritten to use the new index variables.

Conceptually we could calculate, at the beginning of each loop iteration,

\[ I = KU^{-1} \]

and then use this vector \( I \) in the original references, on the form:

\[ x[IA + a_0] \]

We don’t do that of course and instead replace each reference with

\[ x[KU^{-1}A + a_0] \]

Here \( KU^{-1}A + a_0 \) can be calculated at compile-time.
The SUIF compiler was developed at Stanford University.
The project was lead by Monica Lam — who also invented modulo scheduling for software pipelining.
Her project demonstrated that good parallelizing compilers not only must identify parallel loops, but at the same time perform cache optimizations to reduce communication between processors.
Due to synchronization and communication overhead, large computations must be identified that can execute in parallel.

However, only identifying (or creating through transformations) parallelism is not sufficient.

Early parallelizing compilers were not very good at optimizing both for:

- parallelism, and
- locality

at the same time.

This limited their success significantly.

Due to shared memory and multiple levels of caches, multiprocessors make this complex.
As the SUIF compiler project showed, both parallelism and locality must be optimized.

Without such compilers which can manage both, programmers must do it manually, which is expensive and time consuming.

Furthermore, optimizing manually usually means tuning for a particular machine with its cache parameters.

While performance can be good for a specific machine, there is a high risk the performance is not portable.

The tuning needs to be repeated for other machines.

That is another reason for using the best available parallelizing compiler.
Finding Coarse-Grain Parallelism

- Scalar analyses
- Array analyses
- Interprocedural analysis framework
Scalar Analyses

- Privatization of scalar variables
  - each thread gets its own variable
  - in addition to loop index variables, typically also ”temporary” variables used in loops which are defined and used in only one loop iteration

- Scalar reduction recognition (such as \( \text{sum } += \ a[i] \))

- Constant propagation.

- Induction variable recognition and elimination.

- Motion of loop-invariant expressions out of loops.
Perform data dependence tests as explained earlier.

The Fourier-Motzkin test is the most expensive and is performed last if the others fail to prove independence.

SUIF can also privatize arrays.

Recognition of reductions on array elements, i.e. parallel reductions.

Some such reductions are even recognized on the form \( a[b[i]] \).

The latter is useful for sparse computations.
Instead of the simpler solution of inlining is interprocedural dataflow analysis performed.

Inlining does not scale to large programs and should not be used instead of proper interprocedural dataflow analysis.

The dataflow information is expressed as a (mathematical) function of the (source code) function’s arguments.

When the calling contexts are sufficiently different, the framework selectively clones the called procedure, i.e. makes a copy which is tuned to the specific calling context.

This approach gathers as much specific information as does inlining but consumes much less memory — both in the compiler and compiled executable.

In the FORTRAN 77 application TURB3D from SPEC CPU95, loops with up to nine functions in 42 calls, and (if inlined) 86,000 lines were parallelized.

The complete TURB3D benchmark is slightly more than 2000 lines.
Memory Optimization Issues

- Communication — true sharing misses
- Limited capacity — numerical applications often access huge amounts of data before reusing the same data which results in poor temporal locality.
- Limited associativity — if the data is mapped to the same cache locations there can be conflict misses
- Large cache block size — risks of false sharing misses
- The SUIF compiler tries to avoid these issues as explained shortly, and it attempts to hide the latency of the remaining misses through data prefetching.
Transformations

- The SUIF compiler analyses which parts of a large array each processor will access.
- For example, a 2D array where a processor accesses every $n$th row can be transformed into a 3D array where all accessed rows are contiguous in memory.
- Arrays can be transposed when that increases locality.
- By making a processor’s data contiguous in memory, both true and false sharing misses are reduced.
The SUIF compiler set a world record for SPEC CPUfp95 using a machine with eight Alpha processors.

Two of the benchmarks resulted in speedups of approximately 10 and 15 times. How can that happen?

Of 18 applications, interprocedural analysis was essential for seven, and locality optimizations for three.
Since there are so many important sequential programs which need performance improvements, new approaches are possibly required.

One proposed solution is to let the hardware guess cleverly where an independent thread might exist in the stream of instructions, run it speculatively and terminate it when an error is detected.

The shared cache is the ideal "police" here: it can see which thread uses which data and in which order.

There is one non-speculative thread and a number of speculative threads running ahead.

If a speculative thread reads something which the original thread writes later, then a violation error is detected.

Similar violations can happen for write/read and write/write.
In a multithreaded program (or compiler-parallelized code) the software must have explicit synchronization primitives.

In a hardware-based thread speculation system, the hardware runs new threads and if there is no data dependence violation, everything was fine.

So one obvious question is which is fastest?

It depends, of course, on the amount of dependence violations.

Either can be fastest.

Programmer directives can tell the hardware where speculation is most promising — typically derived from a profiler.
Finding Threads

- Two issues: memory dependence violations and the same for registers.
- In Stanford Hydra, the hardware only considers memory dependences.
- This means every thread must access memory for a variable at least once so a violation can be detected.
- Two reasonable places for starting a thread: loop iteration and function call.
- In a call, the original thread does the call and a speculative thread continues after the call in the calling function.
- Hydra schedules in hardware the four least speculative threads (including the non-speculative thread).
- Eventually, a speculative thread either is killed or becomes the oldest.
Requirements on the Memory System in Hydra

- Forward data between parallel threads.
- Detect when a read occurs too early.
- Safely discard speculative state after violations.
- Retire speculative threads in the correct order.
- Provide memory renaming.

We will next see what each means.
Forward Data Between Parallel Threads

- A write by a thread invalidates the copies in future thread’s caches (so that they can safely fetch the data from the shared cache if needed).
- A write does not invalidate earlier threads’ caches but a certain bit is set there anyway to deal with the case when a completely new thread will run on the earlier thread’s processor. That new thread should see the invalidation.
- The shared cache contains the permanent state and each thread has a write buffer at the shared cache, where speculative writes are stored.
- When a cache block is fetched from the shared cache, data in earlier threads’ write buffers are merged with the data in the shared cache.
Detect when Reads Occur Too Early

- Recall that when a write occurs, future threads’ cache copies are invalidated.
- We want to detect that a future thread has read something in the wrong order.
- By noting in future thread’s cache which words have been read, then when an invalidation comes, it can be checked whether the future thread has violated a data dependence. If so, the future thread is restarted.
The permanent state is in the shared cache.

A future thread is not allowed to update the shared cache before it becomes non-speculative.

Any thread writes to its L1 data cache.

Therefore, while the Hydra uses write-through L1 caches, future threads write to a buffer at the shared cache.
Recall all speculative writes are done to buffers.

Simply transferring data from the buffers to the shared cache in the correct order, solves the problem.
A processor can only read data written by itself or earlier threads.

A write from a future thread should not invalidate a cache line in a L1 data cache.

But, when the current thread running on a processor terminates and an even newer thread is run there instead, then that new thread should see the invalidations.

Therefore, the pre-invalidate bit mentioned before is needed to hold information about an invalidation from a future thread and delay the invalidation.
Performance of Stanford Hydra

- Some programs see speedups of 2-3 times.
- With feedback information, the programmer could fix some problems and many programs saw speedups above 1.5.
- They use special syntax e.g. pwhile to make a parallel while loop when the programmer should control the parallelization — and not the hardware.
- Such marked loops will cause the compiler to make certain variables private to each thread.
Data centers running databases and web servers have two big problems:
- To achieve very high throughput so that no requests are blocked, and
- the power consumption of the machines (multiprocessors).

Throughput is perhaps more difficult. What is needed in conventional machines for throughput is high performance for each parallel thread.

Since many requests are independent of each other, it is often very easy to run the server on a multiprocessor with good speedups.

The problems of ever-more complex superscalar processor and longer-latency cache misses do not solve the throughput problem.
Normal multiprocessors are ideal for high-performance computations.

Web servers typically don’t need extremely high performance single threads but rather high throughput.

As has been discussed in industry and academia for decades, one interesting architecture is multithreaded processors.

While one thread in the pipeline is waiting for memory, hardware can schedule another thread immediately (e.g. the next clock cycle) to do useful work.

In fact, web servers and multithreaded processors are a very good match.
Olukotun co-founded a start-up company, Afari Websystems Inc.
The goal was to build server machines powered by multithreaded chip multiprocessors based on the SPARC architecture.
After 9/11 the investor climate was not so good — in fact Olukotun was going to a business meeting in WTC that morning but late enough to avoid being murdered.
Sun bought the start-up.
Kunle Olukotun then spent time at Sun to develop the Niagara processor there.
Niagara Goals versus the Competition

- High throughput using multiple hardware threads and a crossbar to get very high bandwidth in the connect between the processors and a shared L2 cache.

- Lower power requirements than today’s machines: e.g. Google need 400-700 W/sq foot while typical data centers support 70-150 W/sq foot.

- Commercial server applications often have little instruction level parallelism and low cache hit rates, which means the power consumption made by superscalar processors typically is not worthwhile.
The Niagara processor has 8 pipelines where each pipeline is shared by 4 threads.
The shared cache is 3 MB and 12-way set associative.
The crossbar provides 200 GB/s bandwidth.
Each thread has its own store buffers and registers.
Each pipeline is single-issue and has a Thread select in addition to the classic Fetch, Decode, Execute, Memory, and Write Back.
Reminds us of early 1980s pipelines — except for thread select...
Energy consumption by a Niagara processor is 74 W at 1.4 GHz.
Compare it though with a complete PS3 which consumes about 95 W at 3.2 GHz (measured with a tool from Kjell & Co).
More Pipeline Details

- A long latency instruction, e.g. mul or div, causes a thread switch.
- Structural hazards (e.g. two threads that both need the divider) cause one of them to wait.
- Default is to select the least recently selected thread each cycle.
- A thread which wants to issue a load (i.e. something which might miss) is given lower priority than a thread with e.g. an add.
- The register file has three read ports and two write ports (for normal write and for e.g. divide that completes).
Simultaneous Multithreading: SMT

- Susan Eggers and Joel Emer invented Simultaneous Multithreading
- It uses a normal superscalar core
- Multiple hardware threads share this core: pipeline and cache
- This is more flexible than Niagara since compute intensive applications can exploit the pipeline
- Servers can use multiple threads
- POWER8 has 1, 2, 4, or 8 hardware threads per core
A transaction is a sequence of reads and writes which either occur atomically or not at all.

Consider the swish lab:
- 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 transaction
- the end of a transaction

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`
  ```c
  __transaction_atomic {
    from->balance -= amount;
    to->balance += amount;
  }
  ```
- Possibly coming to C++ in the future
Software defines the transactions (e.g. one loop iteration).
Transaction 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
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

- 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 false conflicts!
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 bug 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.
  - Avoid prohibition of 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
  - more difficult to set break points in the middle of a transaction
- Privatization:
  - trivial with locks
  - TM needs hardware support or performance penalty
Tom Knight

- Studied at MIT partly 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
- Took courses as a student at MIT on biology
- Founder of the field synthetic biology
- Published a paper 1986 about transactional memory for Lisp
- Suitable that you will use Clojure for transactional memory
Sun’s Rock processor

- It 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 create the COMA machine there)
- It was produced as prototypes but canceled by Larry Ellison
- In the 2008 financial crises many of Sun’s customers went out of business
- Oracle bought Sun
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.
- TM is implemented partly by having a version of each memory block.
- 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.
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 using L2 caches and the cache coherence protocol
- The L2 cache can store different versions of the same cache line
- Different threads use different versions
- Accesses are marked as reads or writes
- Accesses are also marked as speculative or not
- For speculative accesses it is in addition marked which other threads have accessed the line
- Conflicts are detected using this information in the L2 cache
- The normal granularity is 64 byte
- Two modes: short and long transactions
- They cannot be mixed
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 line is evicted (removed from the L1 cache, and in this case moved to the L2 cache since it was just modified)
- After a store, subsequent loads get L1 cache misses
- This is expected to work OK for short transactions which do not reuse speculative data very much
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.
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 table for position independent code
  - the instruction address for the register restore code
  - a copy of the time base register (see `timebase.c` in Lab 3 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

How does the compiler determine this?
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
Ensuring progress

- 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 **irrecoverable token** which is a global lock
- A completed irrevocable transaction (as identified by its instruction address) may at runtime be marked as **problematic**.
- Problematic transactions switch to irrevocable after first failure
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 exjobb reports
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” :)
See what happens and use that to explain the numbers
L1 cache misses

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># L1 misses per 100 instr. (thread=1)</th>
<th>Instr. path length relative to serial (thread=1)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sequential</td>
<td>BG/Q Short</td>
</tr>
<tr>
<td>bayes</td>
<td>0.6</td>
<td>8.1</td>
</tr>
<tr>
<td>genome</td>
<td>0.6</td>
<td>1.6</td>
</tr>
<tr>
<td>intruder</td>
<td>0.8</td>
<td>1.5</td>
</tr>
<tr>
<td>kmeans_low</td>
<td>0.1</td>
<td>0.3</td>
</tr>
<tr>
<td>kmeans_high</td>
<td>0.1</td>
<td>0.7</td>
</tr>
<tr>
<td>labyrinth</td>
<td>0.9</td>
<td>1.0</td>
</tr>
<tr>
<td>ssca2</td>
<td>1.0</td>
<td>0.7</td>
</tr>
<tr>
<td>vacation_low</td>
<td>1.5</td>
<td>6.2</td>
</tr>
<tr>
<td>vacation_high</td>
<td>1.7</td>
<td>7.0</td>
</tr>
<tr>
<td>yada</td>
<td>1.5</td>
<td>7.2</td>
</tr>
</tbody>
</table>

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