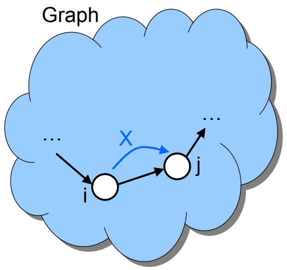
Boundless Transactional Memory

C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul,
Charles E. Leiserson, Sean Lie

MIT Computer Science and Artificial Intelligence Laboratory

Boston Area Computer Architecture Workshop
Friday January 30th, 2004

A Parallel Program Example



```
...
Push-relabel {
...
Flow[i] = Flow[i] – X;
Flow[j] = Flow[j] + X;
...
}
...
```

- We will use the parallel Push-Relabel Max-Flow algorithm to illustrate synchronization.

Achieving Atomicity with Locks

- Correct parallel execution requires atomic regions.

Serial Code

```
Flow[i] = Flow[i] – X;
Flow[j] = Flow[j] + X;
```

Parallel Code using Locks

```
int a, b;
if (i<j) {
  a = i;
  b = j;
} else {
  a = j;
  b = i;
}
Lock(L[a]);
Lock(L[b]);
Flow[i] = Flow[i] – X;
Flow[j] = Flow[j] + X;
Unlock(L[b]);
Unlock(L[a]);
```

- Traditionally, locks are used to achieve atomicity but…
- Hard to use
  - □ Deadlock, priority inversion, etc.
- Conservative
  - □ Atomic regions protected by the same lock can never run concurrently.
- High overhead
  - □ Deadlock avoidance.
  - □ Instructions used in locking are slow.
  - □ Locks use memory.

Achieving Atomicity with Transactions

- Ideally, we want atomicity without those problems.

Serial Code

```
Flow[i] = Flow[i] – X;
Flow[j] = Flow[j] + X;
```

Parallel Code using Locks

```
int a, b;
if (i<j) {
  a = i;
  b = j;
} else {
  a = j;
  b = i;
}
Lock(L[a]);
Lock(L[b]);
Flow[i] = Flow[i] – X;
Flow[j] = Flow[j] + X;
Unlock(L[b]);
Unlock(L[a]);
```

Parallel Code using Transactions

```
XBEGIN;
Flow[i] = Flow[i] – X;
Flow[j] = Flow[j] + X;
XEND;
```

The Case for Hardware Transactions

- Transactions achieve atomicity without those problems.

- Easy to use
  - The hardware simply guarantees transactions are atomic.
- Optimistic
  - Transactions can run concurrently when no memory operations conflict.
  - Easy to implement using much of the processor’s speculation hardware.
- Low overhead
  - No need for deadlock avoidance
  - Fast execution can be achieved using cache and cache-coherency effectively.
  - No lock memory overhead.

Parallel Code using Transactions

```
XBEGIN;
Flow[i] = Flow[i] – X;
Flow[j] = Flow[j] + X;
XEND;
```

The Case for Boundless Transactions

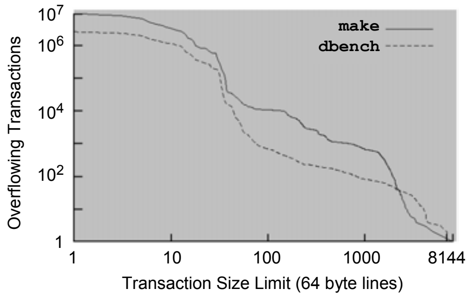
- Previous hardware transaction schemes enforced a bound on transaction size but…

- Exposing a transaction size limit is awkward.
- Dealing with transaction size limit in the compiler is a nontrivial task.
- In typical applications, most transactions are really small but there are some that are huge.

The Case for Boundless Transactions Evidence from the Linux Kernel

- 99.9% of transactions fit in 54 cache lines.
- The largest transaction is more than 7000 cache lines.

Transaction Size Distribution in the Linux Kernel



Boundless Transactional Memory

Goal:

- Run small transactions fast!
- Large transactions can be slower but must still work.

Transactional Memory ISA

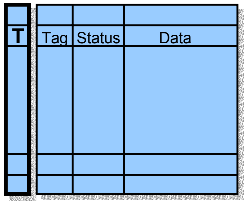
- Add `XBEGIN` and `XEND` instructions to the ISA.

```
Trans_begin:
  XBEGIN abort_handler
  LW R1, 0(R2)
  ADDI R1, R1, #1
  ...
  XEND
  ...
Abort_handler:
  ...           // Back-off
  J Trans_begin // Retry
```

- Transaction = all instructions between `XBEGIN` and `XEND`
- If a transaction completes, the hardware ensures that it is atomic.
- If a transaction cannot complete atomically, it is aborted.
- On an abort, the processor jumps to the abort handler and all state is rolled-back to the `XBEGIN`.
- In the abort handler, the program can back-off and retry.

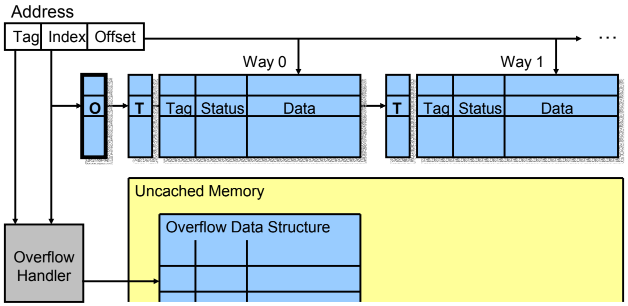
Transactional Data Storage

- All transactional data is stored in the L2 cache. [Herlihy & Moss, Knight]
- Transactional data is primarily stored in the cache
- 1 additional bit (T) per cache line is used to mark the line as transactional.
- Transactional changes can be "rolled-back" by invalidating the transactional cache lines.



- Transactional changes can be committed by clearing all T bits.
- Using the cache achieves very low overhead.

Transactional Data Overflow



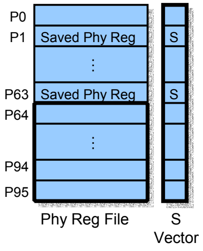
- 1 additional bit (O) per set is used to mark if data has overflowed from the set.
- Overflowed data is stored in an unsorted array in uncached DRAM.
- Overflow data structure is checked on every request hitting an overflowed set.

Transaction Commit and Abort

- Conflicts are detected when an incoming cache intervention hits a transactional cache line.
- Transaction Commit occurs when no conflicts are detected during transaction execution.
  - The T and O bits are cleared. All transactional changes become globally visible.
  - All overflowed cache lines are written back to main memory.
- Transaction Abort occurs when a conflict is detected during transaction execution.
  - All transactional cache lines are invalidated. All transactional changes are discarded.
  - The T and O bits are cleared.
  - All overflowed cache lines are discarded.
  - The processor jumps to the abort handler address given at `XBEGIN`.
  - The processor state is restored to the point just before `XBEGIN`.

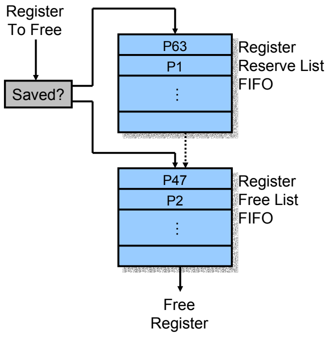
Processor State Save

- Saving the architectural register state of the processor:
- Processor state save is done using much the of the existing speculation hardware in the MIPS R10K.
- 32 physical registers are added to save the 32 architectural registers.
- 1 additional bit per physical register is added to mark it as saved (S).
- A Register Reserve List FIFO is added to store free physical registers that are saved.



Processor State Snapshot

- State Save occurs on XBEGIN.
  - □ The rename table and active S vector are saved away.
  - □ Before commit, all physical registers marked in the S vector are freed into the Reserved List instead of the Free List.
  - □ After commit, the Reserved List is allowed to trickle into the Free List.
- State Restore occurs on transaction abort.
  - □ The saved rename table is restored.
  - □ The Reserved List is cleared.



Evaluation

- We implemented this in the UVSIM simulator and measured 1-processor overheads for the SPECjvm98 benchmark suite.
- In all but one case, transaction overhead is much less than lock overhead.
- 213javac runs everything inside one large transaction. The 14x slowdown suggests that a better data structure is necessary.

Overhead
(% of Serial)

Benchmark	Locks	Transactions
200check	124%	102%
202jess	141%	107%
209db	142%	105%
222mpegaudio	100%	100%
228jack	175%	104%
213javac	170%	1470%

Conclusions

- Boundless Transactional Memory is possible with minimal changes to the hardware.
- By amortizing the high cost of large transactions over the fast small transactions, we are able to achieve much lower overheads than locks in most cases.