



Some thoughts on TM Interfaces

Mark Moir, Principal Investigator

Scalable Synchronization Research Group

Sun Microsystems Laboratories

TM involves many interfaces

- Interfaces for programmers
 - > Expedient transactional library interfaces for supporting experimentation (DSTM, DSTM2, TL2, others, ...)
 - > Static multi-location operations (e.g., DCAS, n-CAS)
 - > Wrappers for lock elision (e.g., U Texas)
 - > TM features/extensions for high-level programming languages (C, C++, JavaTM, ...)
- Such interfaces (except maybe first category) are clear candidates for discussion and (eventual) standardisation.

TM involves many interfaces

- Compiler-library interfaces
 - > Intel ABI for STM C++
 - > Sun compiler-library interface
 - > ...
- Less clear whether discussion, agreement, standardisation is necessary
- Still, aids experimentation with modified/alternative components (e.g., runtime library, compiler)
- To be “standard”, must be architecture independent
- To be flexible, avoid premature optimisation

TM involves many interfaces

- Other runtime interfaces
 - > debugger
 - > performance instrumentation
 - > ...
- Yossi proposes interfaces to support debugging for different TM runtimes, different debuggers
- Flexibility, generality, avoidance of duplicated effort
- Worthwhile to discuss and eventually standardise

Hardware support

- Various hardware features proposed, with various interfaces:
 - > Best-effort HTM (BEHTM), as in Sun's Rock
 - > AMD's Advanced Synchronisation Facility (ASF)
 - > Read Set Monitoring (RSM)
 - Intel's support for HASTM
 - Similar mechanism discussed on Dice blog
 - Rochester's Alert On Update (AOU)
 - > Interfaces for supporting mixed hardware-software unbounded TM implementations
 - Stanford “Architectural Semantics” paper, LogTM and friends, UFO-TM (Zilles), to name a few

Interfaces for hardware support

- Discussing and standardising hardware interfaces is both critical and impossible
- Critical:
 - > how can use become widespread if hardware features all have different interfaces?
 - > will lack of cohesion undermine support for hardware features?
- Impossible:
 - > features must be integrated into different ISAs, therefore must be different
 - > corporate secrecy

The usual approach

- Most of code base architecture independent, small pieces machine dependent where necessary
- No need to port entire code base to new platforms
- Example: SolarisTM defines operations such as `atomic_cas_8`, implemented using `cas` on SPARC[®], `cmpxchgb` on x86.
- This works because `cas` and `cmpxchgb` have same “shape”, functionality.
- Can we do this with TM hardware features?

Not so simple for TM features

- Different features have different interfaces, purposes
- No hope to hide them away in simple, isolated machine-dependent code
- Even closely related features differ significantly
 - > e.g., RSM features differ on how interference is discovered (trap vs. lightweight polling)
- Different vendors might add features in different orders; how can software cope?
- Some ideas...

Simple Machine-Independent TM Interface for BEHTM+RSM

```
void MITMI_resetMonitoring()
T MITMI_loadAndMonitor(T* addr)
    // variants for all relevant types T
bool MITMI_readsStillValid()
int MITMI_beginTransaction()
    // return value indicates txl execution (0), or failure reason (>0)
void MITMI_commitTransaction()
```

Best effort approach

- All features “best effort”, so trivial implementations without hardware support are possible
- Such implementations not directly useful, but allow for different hardware features to be adopted and used in different orders
- In this example:
 - > RSM can fail (almost) always; and/or
 - > BEHTM transactions can fail always

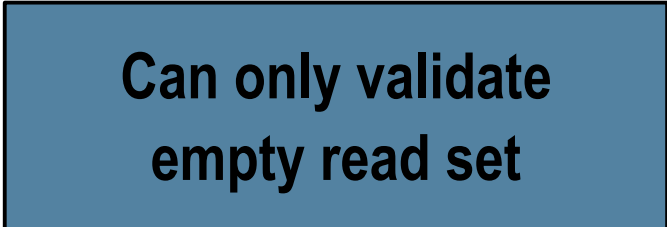
No hardware support for TM (1/2)

```
int tls_anyReadsMonitored = FALSE;
```

```
void MITMI_resetMonitoring() {  
    tls_anyReadsMonitored = FALSE;  
}
```

```
T MITMI_loadAndMonitor(T* addr) {  
    tls_anyReadsMonitored = TRUE;  
    return *addr;  
}
```

```
bool MITMI_readsStillValid() {  
    return !tls_anyReadsMonitored;  
}
```



Can only validate
empty read set

No hardware support for TM (2/2)

```
int MITMI_beginTransaction() {  
    return HTM_FEATURE_NOT_SUPPORTED;  
}
```

All transactions fail

```
int MITMI_commitTransaction() {  
    assert(0);  
}
```

Using Intel's HASTM support (1/2)

```
bool tls_anyReadsMonitored = FALSE;
int  tls_counterSnapshot;
```

```
void MITMI_resetMonitoring() {
    tls_anyReadsMonitored = FALSE;
    HASTM_resetMarkCounter();
}
```

```
T MITMI_loadAndMonitor(T* addr) {
    if (!tls_anyReadsMonitored) {
        tls_anyReadsMonitored = TRUE;
        tls_counterSnapshot = HASTM_readMarkCounter();
    }
    return HASTM_loadSetMark(addr);
}
```



**Take snapshot
at first read**

Using Intel's HASTM support (2/2)

```
bool MITMI_readsStillValid() {  
    return !tls_anyReadsMonitored ||  
           HASTM_readMarkCounter() == tls_counterSnapshot;  
}
```

Use hardware support to
validate monitored reads

```
int MITMI_beginTransaction() {  
    return HTM_FEATURE_NOT_SUPPORTED;  
}
```

```
int MITMI_commitTransaction() {  
    assert(0);  
}
```

Using Rock's BEHTM, but no RSM

// trivial RSM (non)implementation, as before

```
int MITMI_beginTransaction() {  
    ROCK_chkpt failpath  
    return 0;  
failpath:  
    return "failure reason"  
}
```

Start hardware transaction

```
int MITMI_commitTransaction() {  
    ROCK_commit;  
}
```

Commit hardware transaction

Genericising failure feedback

- Rock gives feedback about reasons for transaction failure in special CPS register
- Reasons/feedback are implementation-specific; need to translate to generic reasons: e.g.:

#define REASON_UNKNOWN	1	
#define READ_CONTENTION	2	and/or
#define WRITE_CONTENTION	3	
#define READ_RESOURCES	4	#define NO_ADVICE 1
#define WRITE_RESOURCES	5	#define RETRY 2
#define INSTRUCTION_LIMITATON	6	#define BACKOFF 3
#define EVENT_LIMITATION	7	#define GIVE_UP 4

How might you use this?

- Imagine hardware-assisted STM that can use RSM (if available) to optimise read validation and/or BEHTM (if available) to commit transaction
- Both features best-effort, so need to be able to operate with trivial (non)implementations of one or both, but can take advantage of whatever is available on given platform
- If RSM and BEHTM are both available, we get cheap read validation *and* cheap commit
- Commit transaction must iterate over and validate read set

Integrating best-effort mechanisms

- So far, RSM and BEHTM treated as two separate best-effort mechanisms
- We can change the interface to require them to interoperate (or add variants that do)
- If BEHTM transaction is required to commit only if monitored reads still valid, significant optimizations are possible:
 - > no need to iterate over read set at commit time
 - > no need to even *maintain* read set!
- Integration makes value of the whole greater than sum of values of parts

Using Integrated BEHTM + RSM

// HASTM-based RSM implementation, as before

```
int MITMI_beginRSMTransaction() {  
    HYPO_ROCK_RSM_chkpt failpath  
    return 0;  
failpath:  
    return "failure reason"  
}
```

*Hypothetical variant makes
monitored reads part of transaction*

```
int MITMI_commitTransaction() {  
    HYPO_ROCK_commit();  
}
```

Using Rock's BEHTM, but no RSM

// trivial RSM (non)implementation, as before

```
int MITMI_beginRSMTransaction() {
    if (tls_anyReadsMonitored)
        return RSM_FEATURE_NOT_SUPPORTED;
    ROCK_chkpt failpath
    return 0;
failpath:
    return "failure reason"
}
```

No hardware RSM support, so if any reads monitored, transaction must fail

```
int MITMI_commitTransaction() {
    ROCK_commit;
}
```

Query hardware capabilities

- In previous example, using RSM interface makes MITMI_beginRSMTransaction useless
- Code might infer that, and resort to traditional read set validation and use BEHTM for commit
- Preferable for interface to support querying what capabilities are supported, e.g.:

```
bool MITMI_RSM_support()  
bool MITMI_BEHTM_support()
```

More refined interface could give more information, e.g., resource limitations

What else?

- Could add support for static transactions to interface
- Examples include DCAS, n-CAS
- Would allow support by AMD ASF or Rock's BEHTM
- Best-effort interface, or possibly make guarantees for small simple transactions
- Support for unbounded TM systems, e.g. persistent memory metadata (as in Zilles's UFO bits)

Concluding Remarks

- Many interfaces at various levels involved in implementing and using TM
- Some require standardisation, some may benefit, others maybe should not be standardised
- Useful to discuss all anyway, especially in thinking about hardware support
- Introduced key idea of combining and integrating best-effort mechanisms for flexible hardware support
- But examples are simplistic, not thought out in detail, may have wrong set of features
- Hope to provoke some discussion



**Learn more at
<http://research.sun.com/scalable>**

Questions?

Mark Moir

mark.moir@sun.com