

# Transactional Memory Evaluation using Apache Webserver

Haggai Eran

November 20, 2008

# Outline

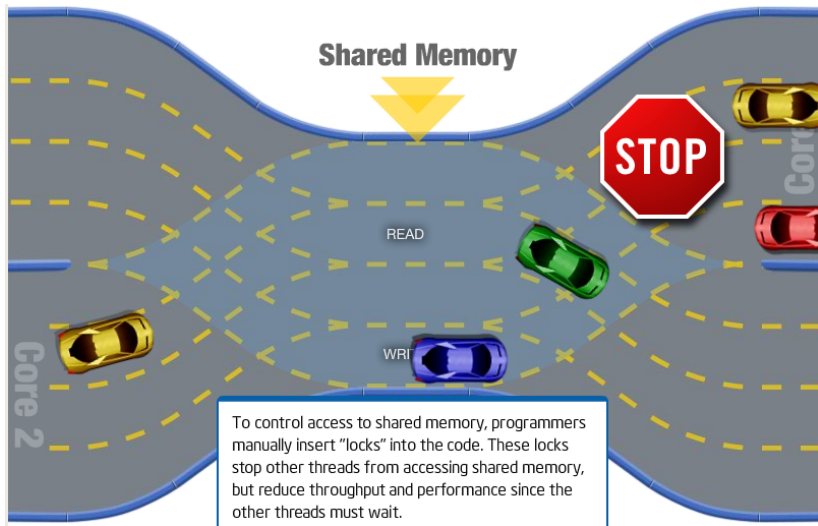
- 1 Transactional Memory
  - Lock based synchronization Limitations
  - Transactional Memory Introduction
- 2 TM Evaluation
  - Evaluation Strategy
  - Transactification Process
  - Evaluation

# Traditional Synchronization

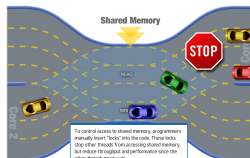
## Example

```
void withdraw(account, amount) {  
    accounts[account] -= amount;  
}
```

# Course-Grained Locks



# Course-Grained Locks



## Example

```
void withdraw(account, amount) {  
    lock(big_mutex);  
    accounts[account] -= amount;  
    release(big_mutex);  
}
```

- Easy to program.
- Doesn't scale.

# Course-Grained Locks

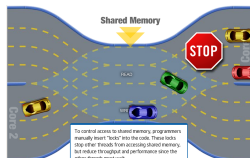


## Example

```
void withdraw(account, amount) {  
    lock(big_mutex);  
    accounts[account] -= amount;  
    release(big_mutex);  
}
```

- Easy to program.
- Doesn't scale.

# Course-Grained Locks

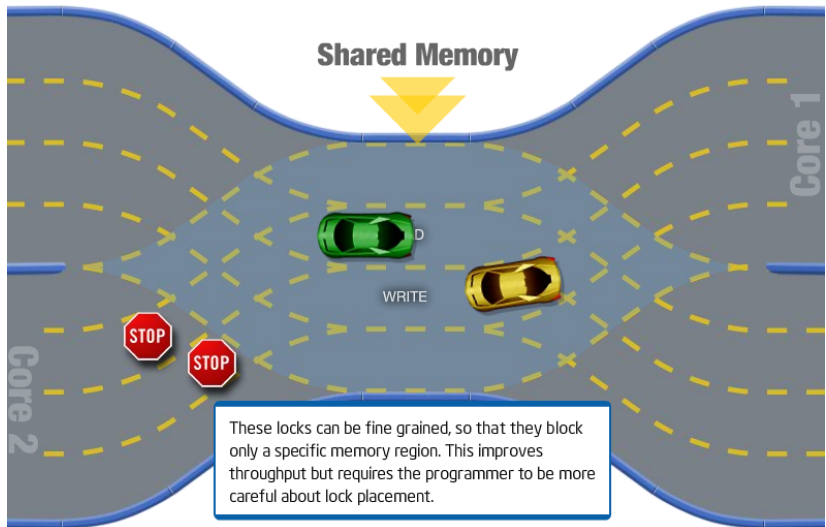


## Example

```
void withdraw(account, amount) {  
    lock(big_mutex);  
    accounts[account] -= amount;  
    release(big_mutex);  
}
```

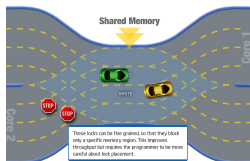
- Easy to program.
- Doesn't scale.

## Fine-Grained Locks





# Fine-Grained Locks

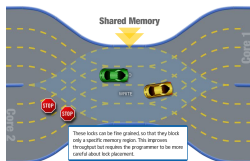


## Example

```
void withdraw(account, amount) {  
    lock(accounts[account].mutex);  
    accounts[account] -= amount;  
    release(accounts[account].mutex);  
}
```

- Can scale well.
- Difficult to program.

# Fine-Grained Locks

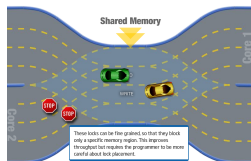


## Example

```
void withdraw(account, amount) {  
    lock(accounts[account].mutex);  
    accounts[account] -= amount;  
    release(accounts[account].mutex);  
}
```

- Can scale well.
- Difficult to program.

# Fine-Grained Locks



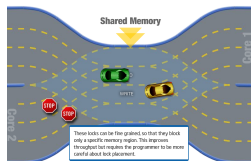
## Example

```
void withdraw(account, amount) {  
    lock(accounts[account].mutex);  
    accounts[account] -= amount;  
    release(accounts[account].mutex);  
}
```

- Can scale well.
- Difficult to program.

# Fine-Grained Locks Difficulties

## Composition



### Example

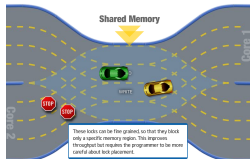
```
void transfer(fromAccount, toAccount, amount) {  
    withdraw(fromAccount, amount);  
    deposit(toAccount, amount);  
}
```

- Locking both accounts from transfer - breaks encapsulation, deadlocks.

• Big lock - decreases performance

# Fine-Grained Locks Difficulties

## Composition



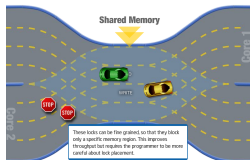
### Example

```
void transfer(fromAccount, toAccount, amount) {  
    withdraw(fromAccount, amount);  
    deposit(toAccount, amount);  
}
```

- Locking both accounts from transfer - breaks encapsulation, deadlocks.

# Fine-Grained Locks Difficulties

## Composition



### Example

```
void transfer(fromAccount, toAccount, amount) {  
    withdraw(fromAccount, amount);  
    deposit(toAccount, amount);  
}
```

- Locking both accounts from transfer - breaks encapsulation, deadlocks.
- Big lock - decreases performance

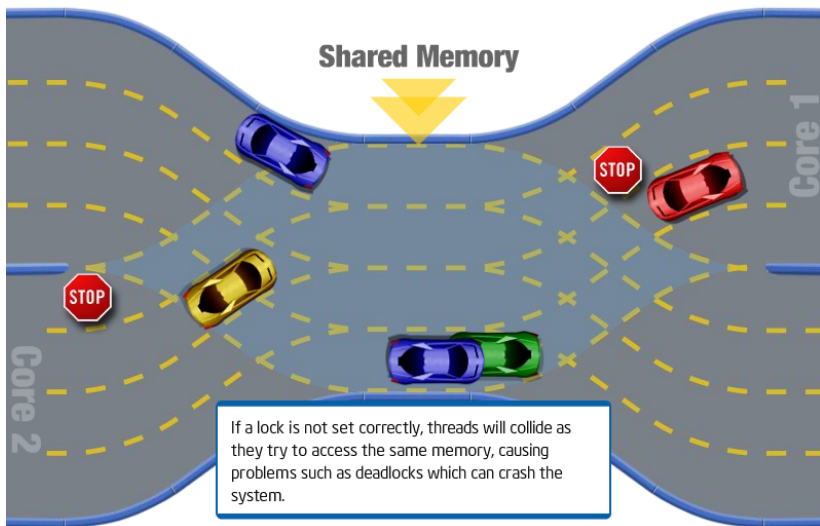
# Fine-Grained Locks Difficulties

## Locking Policies

### Comment from the linux kernel

```
/*  
 * When a locked buffer is visible to the I/O layer  
 * BH_Laundry is set. This means before unlocking  
 * we must clear BH_Laundry, mb() on alpha and then  
 * clear BH_Lock, so no reader can see BH_Laundry set  
 * on an unlocked buffer and then risk to deadlock.  
 */
```

## Fine-Grained Locks Difficulties





# Transactional Memory

- Provide a simple API for programmers.
- Offering fast implementations.

# Transactional Memory

## Simple API

### Example

```
void withdraw(account, amount) {  
    atomic {  
        accounts[account] -= amount;  
    }  
}
```

### Nested transactions

```
void transfer(fromAccount, toAccount, amount) {  
    atomic {  
        withdraw(fromAccount, amount);  
        deposit(toAccount, amount);  
    }  
}
```

# Transactional Memory

## Simple API

### Example

```
void withdraw(account, amount) {  
    atomic {  
        accounts[account] -= amount;  
    }  
}
```

### Nested transactions

```
void transfer(fromAccount, toAccount, amount) {  
    atomic {  
        withdraw(fromAccount, amount);  
        deposit(toAccount, amount);  
    }  
}
```

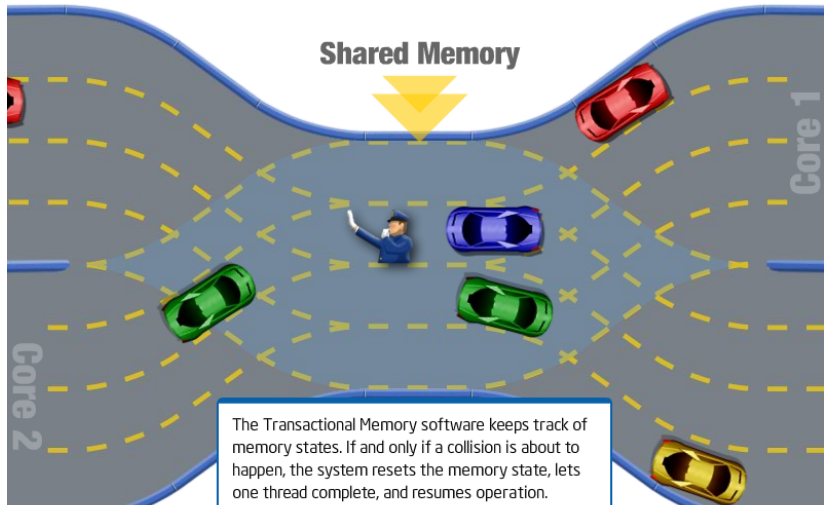


# Transactional Memory

## Implementation

- A transaction is run speculatively without taking any locks.
- Collisions are detected either at commit time or during the run.
- On collision, one of the transactions is aborted and its changes are rolled back.
- Later the aborted transaction is restarted.

# Transactional Memory Implementation



# Transactional Memory

Implementation by software

- All global memory accesses are handled by a special library.
- The library detects collisions and handles commits and aborts.

# Transactional Memory

## Implementation by hardware

- Reuse the cache coherency mechanism in multicore/multiprocessor machines.
- Requires special hardware.
- Limitations: Size and duration of transactions, context switches.

# Outline

- 1 Transactional Memory
  - Lock based synchronization Limitations
  - Transactional Memory Introduction
- 2 TM Evaluation
  - Evaluation Strategy
  - Transactification Process
  - Evaluation



# Existing Benchmarks

- Red-Black trees benchmarks
- STAMP benchmark suite.
  - Bayesian network learning
  - Gene sequencing
  - Network intrusion detection
  - K-means clustering
  - Maze routing
  - Graph kernels
  - Client/server travel reservation system
  - Delaunay mesh refinement

## Our Project's Goal

Create a benchmark based on a real-world application for transactional memory.

# Existing Benchmarks

- Red-Black trees benchmarks
- STAMP benchmark suite.
  - Bayesian network learning
  - Gene sequencing
  - Network intrusion detection
  - K-means clustering
  - Maze routing
  - Graph kernels
  - Client/server travel reservation system
  - Delaunay mesh refinement

## Our Project's Goal

Create a benchmark based on a **Apache web-server** for transactional memory.

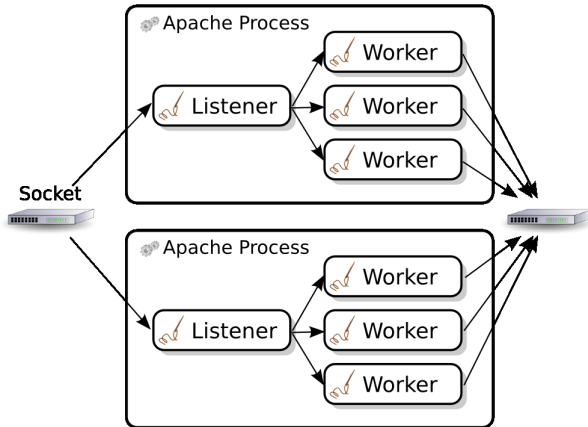
# Apache Web Server



- Written in C.
- Support many Multiprocessing Modules (MPMs): Parallel execution strategies.
- A mainly developed threaded MPM is the Worker MPM: Runs several processes, each running a fixed number of threads.

# Apache Web Server

## Worker MPM



## Apache Cache Module - mod\_\_mem\_\_cache

- There isn't much interaction between the worker threads.
- The cache module enables worker threads of the same process to share cached pages in memory.
- Currently implemented with one big lock.

# Software Transactional Memory in C/C++

Several STM implementations for C are available as libraries.

- Require accessing global variables through library functions / macros.

A few compiler based implementations:

- Tanger - An open-source academic LLVM-based STM compiler.
  - Support using any STM library through a known interface.
- ICC - Intel's experimental STM compiler
  - Works with Intel's own transactional memory manager.

# Software Transactional Memory in C/C++

Several STM implementations for C are available as libraries.

- Require accessing global variables through library functions / macros.

A few compiler based implementations:

- Tanger - An open-source academic LLVM-based STM compiler.
  - Support using any STM library through a known interface.
- ICC - Intel's experimental STM compiler
  - Works with Intel's own transactional memory manager.

# Software Transactional Memory in C/C++

Several STM implementations for C are available as libraries.

- Require accessing global variables through library functions / macros.

A few compiler based implementations:

- Tanger - An open-source academic LLVM-based STM compiler.
  - Support using any STM library through a known interface.
- ICC - Intel's experimental STM compiler
  - Works with Intel's own transactional memory manager.



# Transactifying Compiler

- Modifies code inside atomic blocks to access globals through the STM.
- Function calls.
- Indirect function calls.
- Library functions.

# Transactifying Compiler

- Modifies code inside atomic blocks to access globals through the STM.
- Function calls.
- Indirect function calls.
- Library functions.

# Transactifying Compiler

- Modifies code inside atomic blocks to access globals through the STM.
- Function calls.
- Indirect function calls.
- Library functions.

# Transactifying Compiler

- Modifies code inside atomic blocks to access globals through the STM.
- Function calls.
- Indirect function calls.
- Library functions.

# Commit handlers

A common pattern we found, missing in both Tanger and ICC.

## Example

```
atomic {  
    if (--object.reference_count) {  
        cache_remove(object);  
        destroy(object);  
    }  
}
```

# Commit handlers

Should be converted to:

## Example

```
atomic {  
    if (--object.reference_count == 0) {  
        cache_remove(object);  
    }  
}  
if (object.reference_count == 0)  
    destroy(object);
```

# Commit handlers

It would be nice to have:

## Example

```
atomic {  
  if (--object.reference_count == 0) {  
    cache_remove(object);  
    on_commit(destroy, object);  
  }  
}
```

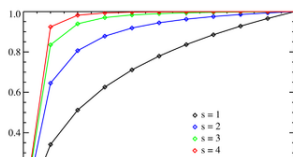
# Evaluation

Evaluation of a web server requires:

- A data set.
- Client strategy

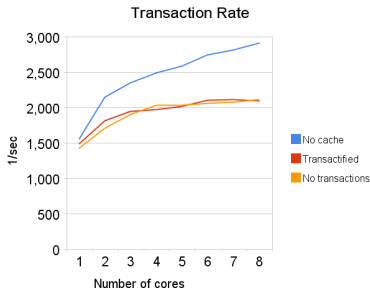
We chose

- Data set of small files (man pages) so that the throughput of the NIC won't be the bottleneck.
- Running as many clients concurrently as possible to create contention on the server and its cache.
- Requesting pages according to Zipf distribution - to control locality.





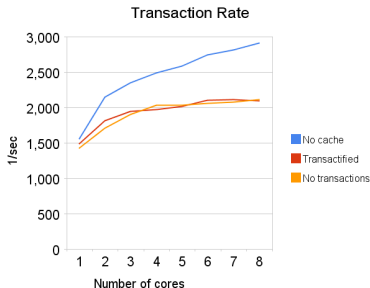
# Current Results



## Theory

- The linux file cache contains the entire data set => Apache's cache just gets in the way.
- Dynamically generated content might give the cache an advantage.

# Current Results



## Theory

- The linux file cache contains the entire data set => Apache's cache just gets in the way.
- Dynamically generated content might give the cache an advantage.

# Thank you

Questions

?