FROM LIGHTWEIGHT HARDWARE TRANSACTIONAL MEMORY TO LIGHTWEIGHT LOCK ELISION

Martin Pohlack, Stephan Diestelhorst
June 5, 2011
Public
LOCK ELISION

- Locking: choice between two evils
  - Coarse grained: Pessimistic, scales badly, simple
  - Fine grained: Might scale, hard to get right

- LE: Reuse critical-section structure in existing parallel software
  - Lock acquisition / release demarcate speculation boundaries
  - Speculative concurrent execution
  - Only real data conflicts serialize
RELATION WORK

- Speculative Lock Elision
  - Ravi Rajwar and James Goodman.: *Speculative lock elision: enabling highly concurrent multithreaded execution.* Micro’01
  - Transparently done in hardware

- Transactional Lock Elision
  - Dave Dice, Yossi Lev, Mark Moir, Dan Nussbaum: *Early Experience with a Commercial Hardware Transactional Memory Implementation.* ASPLOS’09
  - Policy and demarcations in software, ROCK’s HTM

- Adaptive Locks
  - Takayuki Usui, Reimer Behrends, Jacob Evans, Yannis Smaragdakis: *Adaptive Locks: Combining Transactions and Locks for Efficient Concurrency.* PACT '09
  - STM-mode for locks, detailed performance tradeoffs
OUR CONTRIBUTIONS

- Adaptation to AMD’s Advanced Synchronization Facility (ASF)
  - Small change to existing ASF
  - Speculation by Default
  - New instruction: SPECULATE_INV
  - Easy deployment: almost no toolchain support required

- Early results for a memcached-based workload
  - To some, transactional memory is a burned, theoretical concept
  - Business-relevant workload
**ASF IN A NUTSHELL**

- Experimental ISA extension for the AMD64 architecture
- Hardware-transactional-memory proposal
  - Minimally invasive, focus on implementability in high-volume commercial microprocessor
  - Utilizes existing cache design and cache coherence protocols
- Capacity and eventual forward-progress
- Selective annotation: Both transactional and non-transactional accesses supported

ASF Specification

Implemented in out-of-order, full-system timing simulator PTLsim
ASF INSTRUCTIONS

- **SPECULATE**: Start TX, abort reason in RAX, Flags
- **COMMIT**: Conclude TX
- **LOCK MOV**: Transactional memory access (load / store)
- **ABORT**: Manual abort
- **LOCK PREFETCH[W]**: Start monitoring memory line (read / write mode)
- **RELEASE**: Potentially drop memory line from read set
CHANGES TO ASF

- New instruction: **SPECULATE_INV**
  - Changes *polarity* of **LOCK** prefix
  - Everything inside is transactional, except, some types of **LOCK MOV**
  - Coexist with normal **SPECULATE**
  - Might nest with normal **SPECULATE** (not implemented in Simulator)

- Benefits
  - Re-use of existing code, only special entry / exit code required
  - Optimized for transactional code being common case
  - Eases transition
EARLY RESULTS WITH MEMCACHED

- Memcached: what is it?
  - Distributed key-value storage, in-memory DB, cache in front of persistent DB
  - Popular social media sites

- Characteristics
  - Distributed, typically in clusters
  - Each instance multithreaded, synchronized with locks
  - Memory hash table, often short code paths
  - Reads typically dominate workload (GET requests)
EXPERIMENTAL SETUP

- Full-system simulator (ptlsim)
  - Adapted memory latencies to AMD Opteron™ processors of families 0Fh (K8 core) and 10h (formerly code-named “Barcelona”).
  - ASF + SPECULATE_INV
  - Single system: 8 cores (4 + 4)

- memslap benchmark (client)
  - Load testing and benchmarking tool for memcached
  - Typical call: memslap -w10k -c256 -T4 -t500000 -s localhost –B
  - Stand-alone experiments in GB-net environment to find high-throughput parameters
  - Unsuitable: mc-blaster, unstable under high load, ASCII protocol

- mutrace: identify contending locks, helps identifying locks
TRANSACTION CHARACTERISTICS

- Mostly read-only: memslap: 90% reads / 10% writes
- Central hash-table lock
  - Memcached received scalability attention [2, 23, 25, 32]
- Small in terms of instructions / read-set / write-set
  - Actual data not touched with lock held, only retrieved / exchanged via pointer
- Why no RW-Lock?
  - Rare: writing to data structures even in read path (GET request)
  - Resizing of hash table
  - On-access aging of elements (also in read path)
MANUAL INSTRUMENTATION

- Single path and lock
- Small changes to source code required, recompilation
- Static definition / choice of path and lock to instrument
MANUAL INSTRUMENTATION

- Single path and lock
- Small changes to source code required, recompilation
- Static definition / choice of path and lock to instrument

```c
item *item_get(const char *key, const size_t nkey) {
    item *it;
    pthread_mutex_lock(&cache_lock);
    it = do_item_get(key, nkey);
    pthread_mutex_unlock(&cache_lock);
    return it;
}
```
MANUAL INSTRUMENTATION

- Single path and lock
- Small changes to source code required, recompilation
- Static definition / choice of path and lock to instrument

```c
item *item_get(const char *key, const size_t nkey) {
    item *it;  int retries = 5;  ulong asf_fail;

    while (retries > 0) {
        asf_speculate_inv(asf_fail);

        if (cache_lock.__data.__lock) {
            asf_abort(1);
        }
        it = do_item_get(key, nkey);
        asf_commit_();
        return it;
    }

    pthread_mutex_lock(&cache_lock);
    it = do_item_get(key, nkey);
    pthread_mutex_unlock(&cache_lock);
    return it;
}
```
MANUAL INSTRUMENTATION

- Single path and lock
- Small changes to source code required, recompilation
- Static definition / choice of path and lock to instrument

```c
item *item_get(const char *key, const size_t nkey) {
    item *it;  int retries = 5;  ulong asf_fail;

    while (retries > 0) {
        asf_speculate_inv(asf_fail);
        if (UNLIKELY (asf_fail)) {

            retries--;

            continue;
        }
        if (cache_lock.__data.__lock) {
            asf_abort(1);
        }
        it = do_item_get(key, nkey);
        asf_commit_();
        return it;
    }

    pthread_mutex_lock(&cache_lock);
    it = do_item_get(key, nkey);
    pthread_mutex_unlock(&cache_lock);
    return it;
}
```
MANUAL INSTRUMENTATION

- Single path and lock
- Small changes to source code required, recompilation
- Static definition / choice of path and lock to instrument

```c
item *item_get(const char *key, const size_t nkey) {
    item *it;  int retries = 5;  ulong asf_fail;

    while (retries > 0) {
        asf_speculate_inv(asf_fail);
        if (UNLIKELY (asf_fail)) {
            if (asf_hard_error(asf_fail)) {
                retries = 0;
            } else {
                retries--;
            }
            continue;
        }
        if (cache_lock.__data.__lock) {
            asf_abort(1);
        }
        it = do_item_get(key, nkey);
        asf_commit_();
        return it;
    }

    pthread_mutex_lock(&cache_lock);
    it = do_item_get(key, nkey);
    pthread_mutex_unlock(&cache_lock);
    return it;
}
```
DYNAMIC INSTRUMENTATION

- Instrument dynamically
  - LD_PRELOAD
  - Wrap
    pthread_mutex_lock / unlock
- No source code changes, no recompilation
- Lock / path identification done dynamically
DYNAMIC INSTRUMENTATION

- Instrument dynamically
  - LD_PRELOAD
  - Wrap
    `pthread_mutex_lock` / `unlock`
- No source code changes, no recompilation
- Lock / path identification done dynamically

```c
int pthread_mutex_lock(pthread_mutex_t *mutex) {
  int retries; uint64_t asf_fail;

  // care for recursive calls and library loading ...

  return real_pthread_mutex_lock(mutex);
}
```


**DYNAMIC INSTRUMENTATION**

- Instrument dynamically
  - LD_PRELOAD
  - Wrap `pthread_mutex_lock` / `unlock`

- No source code changes, no recompilation

- Lock / path identification done dynamically

```c
int pthread_mutex_lock(pthread_mutex_t *mutex) {
    int retries; uint64_t asf_fail;

    ... care for recursive calls and library loading ...

    // determine retries for mutex and local thread
    retries = elide_retries(mutex);

    return real_pthread_mutex_lock(mutex);
}
```
**DYNAMIC INSTRUMENTATION**

- Instrument dynamically
  - LD_PRELOAD
  - Wrap `pthread_mutex_lock / unlock`
- No source code changes, no recompilation
- Lock / path identification done dynamically

```c
int pthread_mutex_lock(pthread_mutex_t *mutex) {
    int retries; uint64_t asf_fail;

    // determine retries for mutex and local thread
    retries = elide_retries(mutex);
    while (retries > 0) {
        asf_speculate_inv(asf_fail);
        if (mutex->__data.__lock) { asf_abort(1); }
    }
    return 0; // eliding now
}

return real_pthread_mutex_lock(mutex);
```
DYNAMIC INSTRUMENTATION

- Instrument dynamically
  - LD_PRELOAD
  - Wrap
    \[ \text{pthread_mutex_lock / unlock} \]
- No source code changes, no recompilation
- Lock / path identification done dynamically

```c
int pthread_mutex_lock(pthread_mutex_t *mutex) {
    int retries; uint64_t asf_fail;

    ... care for recursive calls and library loading ...

    // determine retries for mutex and local thread
    retries = elide_retries(mutex);
    while (retries > 0) {
        asf_speculate_inv(asf_fail);
        if (UNLIKELY (asf_fail)) {
            if (asf_hard_error(asf_fail)) {
                inc_stat_hard_abort(mutex);
                break;
            } else {
                retries--;
                inc_stat_soft_abort(mutex, retries);
            }
            continue;
        }
        if (mutex->__data.__lock) { asf_abort(1); }
        return 0; // eliding now
    }

    return real_pthread_mutex_lock(mutex);
}
```
**ADAPTATION**

- Statistics per mutex and thread: **Level**
- Increase Level on ASF failure, decrease on ASF success
- Level determines chance to retry with ASF
  - Level 0: 100%, Level 1: 50%, ...
  - Standard vs. Aggressive Level decrease
RESOURCE BREAKDOWN: CACHE LINES
RESOURCE BREAKDOWN: CYCLES

The chart shows a breakdown of cycles, with a red histogram for cycles and a green line representing the cumulative distribution function (CDF). The x-axis represents the number of cycles, ranging from 0 to 250, while the y-axis shows frequency on the left and CDF on the right. The data indicates a distribution of cycles with a peak around 50 cycles, and the CDF curve rises gradually, indicating a concentration of cycles in the lower range.
RESOURCE BREAKDOWN: X86 INSTRUCTIONS

![Graph showing resource breakdown for X86 instructions](image-url)
RESOURCE BREAKDOWN: RETIRED MEMORY INSTRUCTIONS (LD/ST)
## RESULT SNAPSHOT

<table>
<thead>
<tr>
<th>Setup</th>
<th>Throughput (transactions / s)</th>
<th>Improvement (to baseline)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>430 470</td>
<td>0.0%</td>
</tr>
<tr>
<td>Manual instrumentation</td>
<td>524 430</td>
<td>21.8%</td>
</tr>
<tr>
<td>Dynamic instrumentation</td>
<td>559 250</td>
<td>29.9%</td>
</tr>
<tr>
<td>Dynamic instrumentation (aggressive)</td>
<td>576 675</td>
<td>34.0%</td>
</tr>
</tbody>
</table>
OUTLOOK & CONCLUSION

- Other workloads
  - Stray from traditional academic TM workloads
  - GIL-based interpreter languages

- ASF with default-speculation mode
  - Design point with low deployment requirements
  - Suitable for lock elision
  - Low-overhead software approach for policy handling may pay off
The information presented in this document is for information purposes only. The information contained herein is subject to change and may be rendered inaccurate for many reasons, including, but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The research leading to these results has received funding from the European Community’s Seventh Framework Programme (FP7/2007-2013) under grant agreement No 216852.

Trademark Attribution

AMD, the AMD Arrow logo and combinations thereof are trademarks of Advanced Micro Devices, Inc. in the United States and/or other jurisdictions. Other names used in this presentation are for identification purposes only and may be trademarks of their respective owners.

©2011 Advanced Micro Devices, Inc. All rights reserved.