Deterministic Execution

02/27 Ni Kang, Taihai He

DMP: deterministic shared memory multiprocessing

- Summary
- DMP Key Mechanisms
  - DMP-Serial
    - Deterministic Token and Quantum
  - DMP-ShTab
  - DMP-TM/DMP-TMFwd
  - Quantum Builders
- Hardware/Sofeware Implementaions
- Evaluations
- Discussions

Background

- Nondeterminism
  - Same inputs can lead to different outputs
  - Too many possible ways of instruction interleaving
    - “Defective software might execute correctly hundreds of times before a subtle synchronization bug appears, and when it does, developers typically cannot reproduce it during debugging.“
  - Need to use logs to record every execution
    - Still hard to replay

Summary

- Determinism
    - Key: deterministic inter-thread communication
        - Maintain a fixed order of load/store operations on shared data
    - Rest of the instructions can still have different orders in exectuions
        - “Communication-equivalent interleavings”
- Use deterministic execution to improve reliability
    - Easier to test and debug
        - Avoid subtle multithread bugs
        - Always able to reproduce previous execution results
    - Acceptable performance loss
        - Multiple co-existable mechanisms for different applications
        - Complexity-performance trade-offs between hardware and software implementations

Nondeterminism Quantification

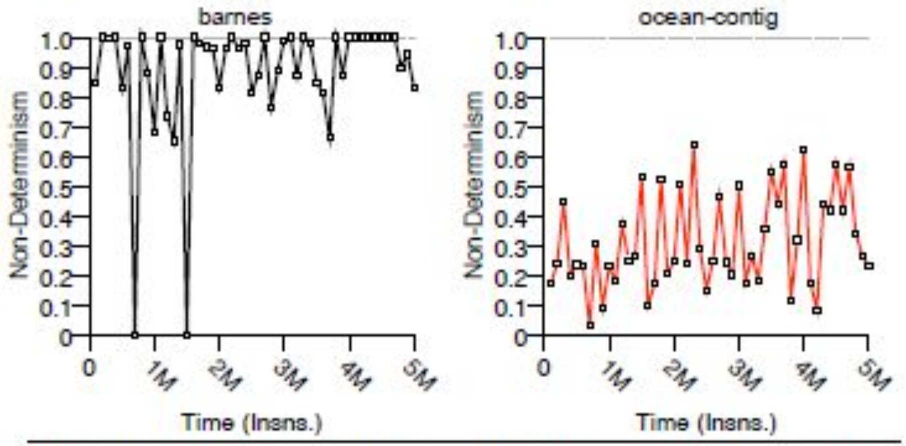


Figure 3. Amount of nondeterminism over the execution of *barnes* and *ocean-contig*. The x axis is the position in the execution where each sample of 100,000 instructions was taken. The y axis is ND (Eq. 1) computed for each sample in the execution.

- Exist regions where nondeterminism drops to nearly zero.
- Executions may never reach 100% nondeterminism.

DMP-Serial

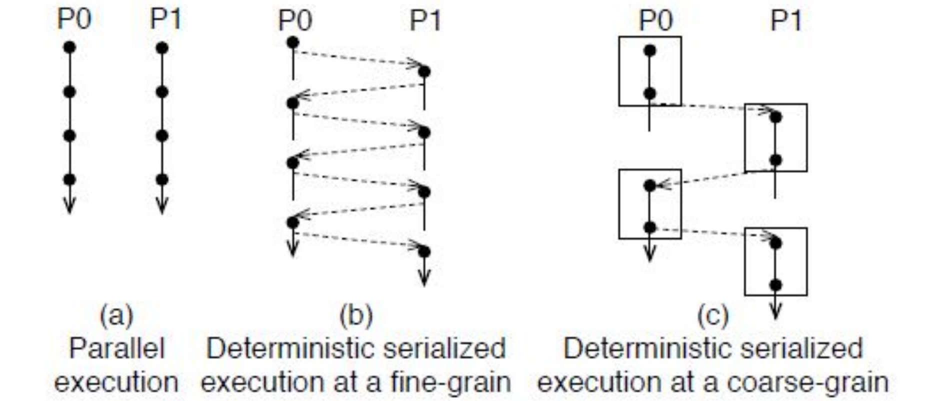


Figure 4. Deterministic serialization of memory operations. Dots are memory operations and dashed arrows are happens-before synchronization.

- DMP-Serial
    - Fully serialized accesses to data
    - Allow only one preocessor at a time to access memeory in a deterministic order

Deterministic Token and Quantum

- Deterministic token
    - Processor with the token can access memory. Otherwise, wait for it.
    - One token passing around. Multiple tokens are also allowed with hardware implementation if there are multiple deterministic processes at the same time
- Quantum
    - Instruction segments invloving shared data load/store that require token
    - QB-Count: count instructions and break when a deterministic, target number is reached
    - Other smarter ways to divide quantum, will introduce later
- Parallelism?
    - Serilization hurts performance a lot

DMP-ShTab

- Not all load/store operations have conflicts
  - Communication is the key
  - Quantum = communication-free prefix + serial suffix
  - Only requires suffixies to be deterministic
- Sharing table for memory locations
  - Data is either private or shared for a processor
  - Supports different granularities
- Features
  - Token is only required for accessing shared data
  - If one thread wants to write data, it needs to wait for all other threads to be blocked even if it has already acquired the token. (Broadcast)
    - Block: finish execution of quantum or prefix

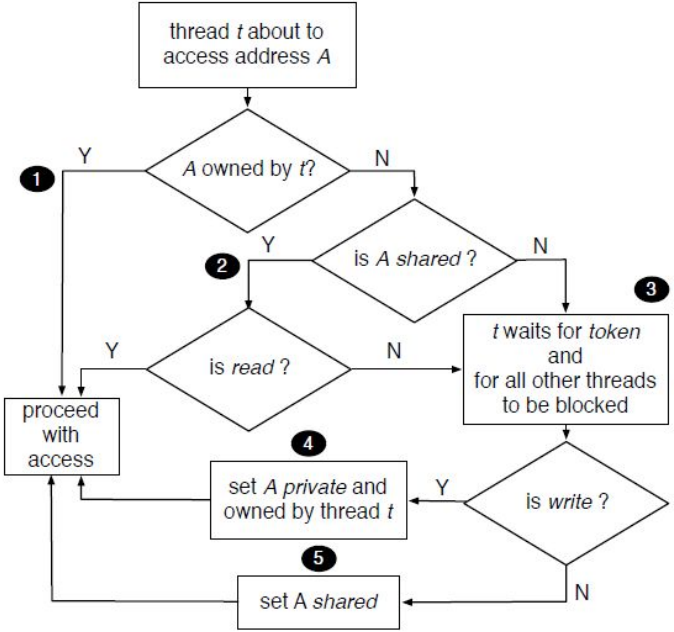


Figure 6. Deterministic serialization of shared memory communication only.

DMP-TM / DMP-TMFwd

- Transactional Memory Support
  - Allowing more concurrent executions with speculations and re-executions
- DMP-TM
  - Speculation + Commit + Squash
  - Correctness: no overlapping memory accesses
  - May squash and re-execute quantum when deterministic serialization is violated
- DMP-TMFwd
  - DMP-TM + Forward
  - Quantum can fetch uncommitted data from other quantum
  - Avoid some squashes, but all subsequent quantum need to be squashed if previous speculations generated incorrect data

Quantum Builders

- A fixed number of instructions may not reflect the progress of a thread on its critical path of execution
- QB-SyncFollow
    - Ends a quantum when an unlock operation is performed
    - Other threads may be waiting for the lock right now
- QB-Sharing
    - Ends a quantum when a thread hasn't issued memory operations to shared locations in some time, like after a number of instructions
    - Other threads don't need to keep waiting if current thread has already finished all of its memory-sensitive operations
- QB-SyncSharing
    - QB-SyncFollow OR QB-Sharing, whenever either of their requirements are satisfied

Hardware / Software Implementation

- Hardware: more complex, better performance (less performance drop)
    - Quantum Building: may need supports from compilers
    - DMP-ShTab
        - Uses MESI cache coherence protocol to represent private / shared status
        - State changing requirements: no speculation, must have token, all threads blocked
        - Similar to directory-based cache coherence
    - DMP-TM / DMP-TMFwd
        - Allowing commit only when token is held
        - Data versioning
        - Similar to Thread-Level Speculation (TLS)
- Software: simple, helpful at debugging-level
    - Use compiler or binary writer
    - Build quantum with CFG
    - Token = lock

Evaluation: mechanisms

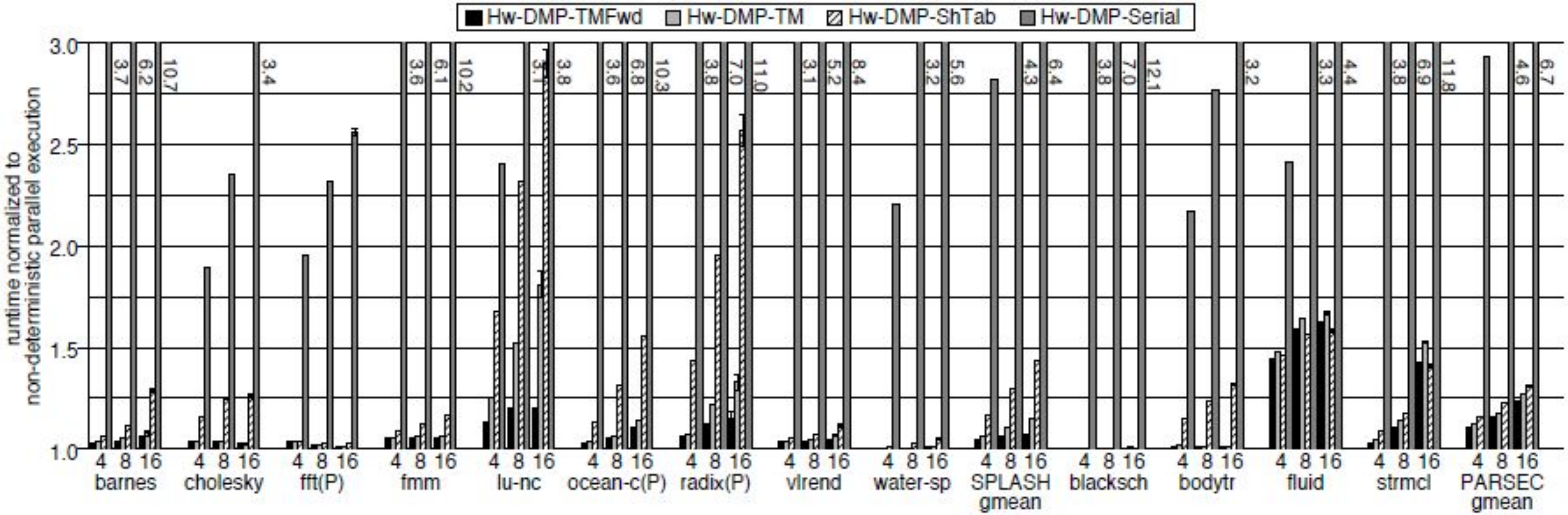


Figure 9. Runtime overheads with 4, 8 and 16 threads. (P) indicates page-level conflict detection; line-level otherwise.

Evaluation: quantum size

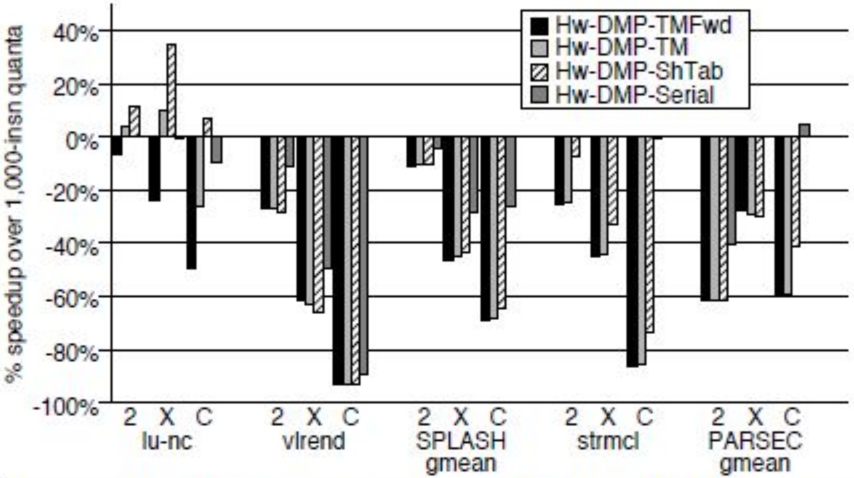


Figure 10. Performance of 2,000 (2), 10,000 (X) and 100,000 (C) instruction quanta, relative to 1,000 instruction quanta.

Evaluation: granularity

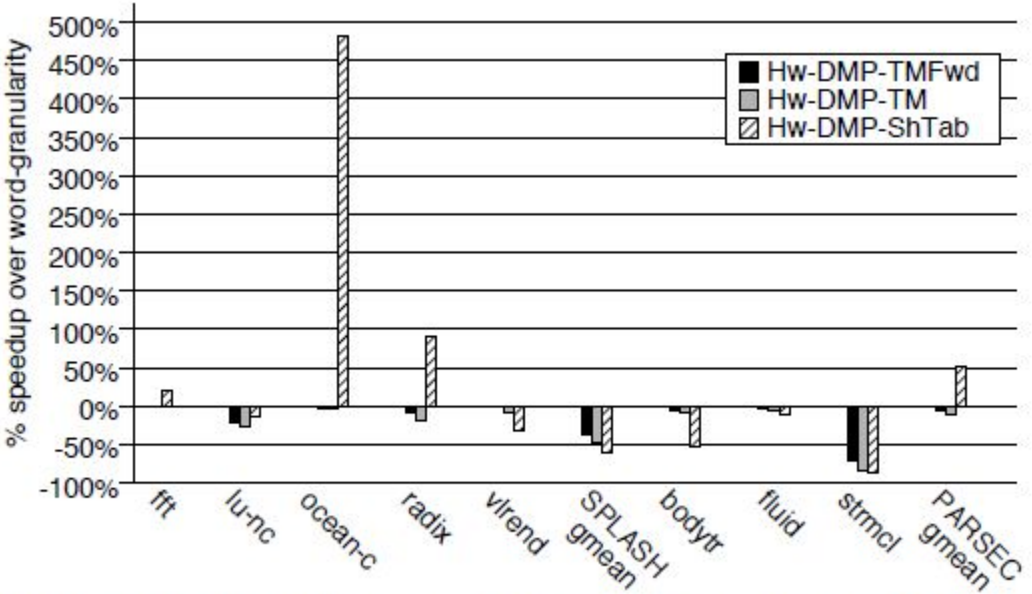


Figure 11. Performance of page-granularity conflict detection, relative to line-granularity.

Evaluation: quantum builders

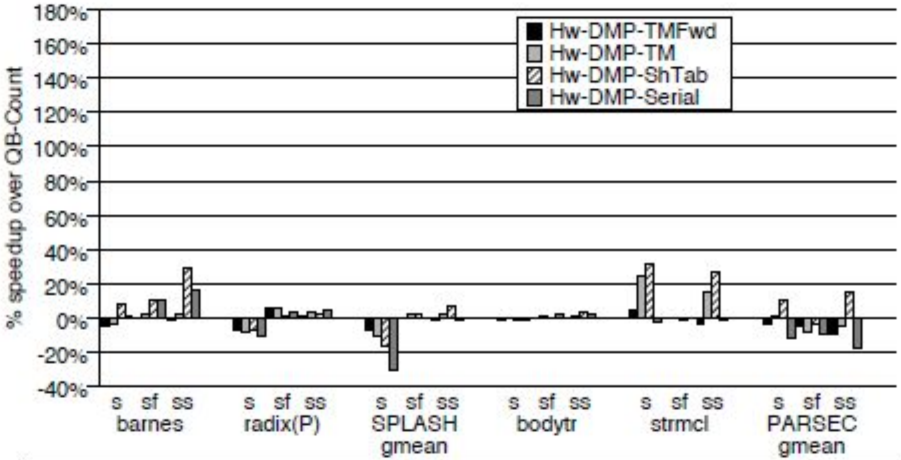


Figure 12. Performance of QB-Sharing (s), QB-SyncFollow (sf) and QB-SyncSharing (ss) quantum builders, relative to QB-Count, with 1,000-insn quanta.

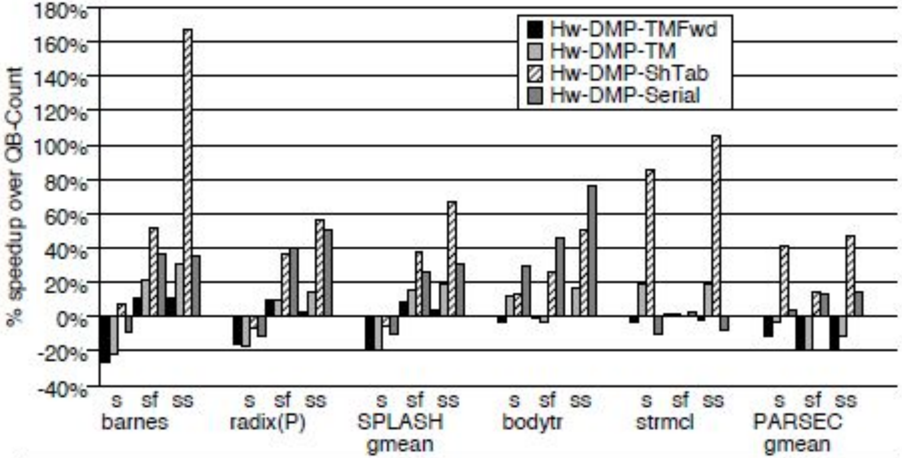


Figure 13. Performance of quantum building schemes, relative to QB-Count, with 10,000-insn quanta.

Evaluation

Benchmark	Hw-DMP Implementation – 1,000-insn quanta: TM Line R/W set sz.	TM Line % conf.	TM Page R/W set sz.	TM Page % conf.	ShTab Line % Q overlap	ShTab Page % Q overlap	QB Strategy – 10,000-insn quanta†: SyncFollow Avg. Q sz.	SyncFollow % sync brk.	Sharing Avg. Q sz.	Sharing % shr. brk.	SyncSharing Avg. Q sz.	SyncSharing % sync brk.
barnes	27/9	37	9/2	64	47	46	5929	42	4658	67	5288	54
cholesky	14/6	23	3/1	39	31	38	6972	30	3189	94	6788	35
fft	22/16	25	3/4	26	19	39	9822	1	3640	62	4677	49
fmm	30/6	51	7/1	69	33	29	8677	15	4465	65	5615	50
lu-nc	47/33	71	6/4	77	14	16	7616	24	6822	37	6060	42
ocean-c	46/15	28	5/2	34	5	46	5396	49	3398	73	3255	73
radix	16/20	7	3/7	13	31	42	8808	15	3346	71	4837	57
vlrend	27/8	38	7/1	50	41	39	7506	28	7005	45	6934	38
water-sp	32/19	19	5/1	45	40	37	7198	5	5617	30	6336	20
SPLASH amean	30/16	31	5/2	44	29	35	7209	27	4987	57	5363	48
blacksch	28/9	8	14/1	10	48	48	10006	<1	9163	10	9488	7
bodytr	11/4	16	3/2	28	39	19	7979	25	7235	31	6519	37
fluid	41/8	76	8/2	75	43	40	871	98	2481	95	832	99
strmcl	36/5	28	10/2	91	60	12	9893	1	1747	79	2998	77
PARSEC amean	29/6	36	9/1	51	45	30	7228	19	5156	54	3880	64

Table 1. Characterization of hardware-DMP results. † Same granularity as used in Figure 9

Evaluation: software implmentation

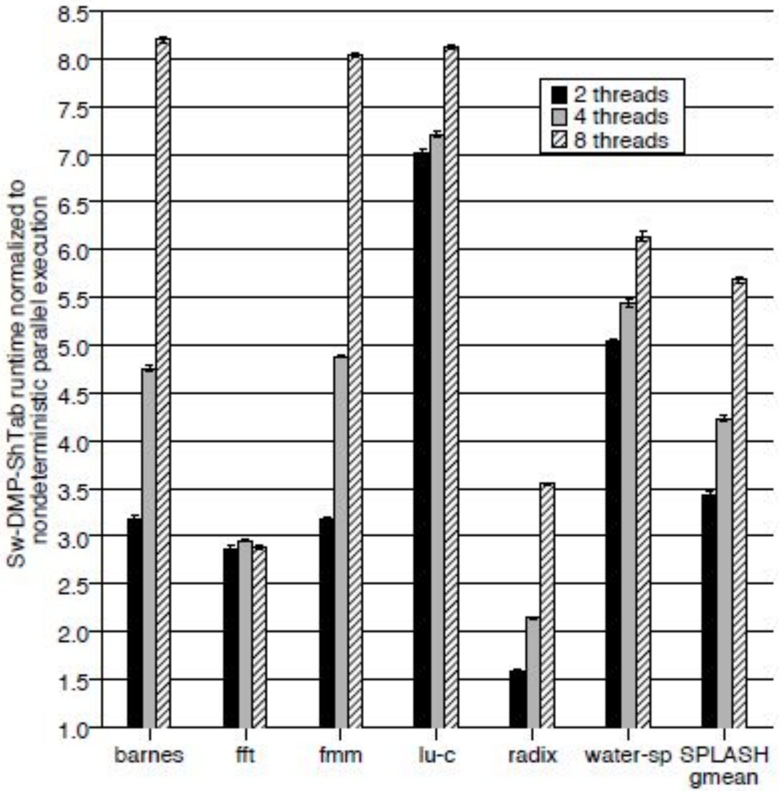


Figure 14. Runtime of Sw-DMP-ShTab relative to nondeterministic execution.

Discussions

- A system can have DMP-TM(Fwd) / DMP-ShTab / DMP-Serial at the same time and switch to each other for different tasks
- Hardware and software implmentations can be used together to have fliexiblity
- Supports deployment with modification and standardization

Grace: Safe Multithreaded Programming for C/C++

Motivation

- Concurrency bugs

Concurrency Error	Cause	Prevention by Grace
Deadlock	cyclic lock acquisition	locks converted to no-ops
Race condition	unguarded updates	all updates committed deterministically
Atomicity violation	unguarded, interleaved updates	threads run atomically
Order violation	threads scheduled in unexpected order	threads execute in program order

Table 1. The concurrency errors that Grace addresses, their causes, and how Grace eliminates them.

Motivation

- Transactional memory system is not working here
  - Compatibility with C/C++ and commodity hardware
  - Support for long-lived transactions
  - Isolation of updates from other threads
  - Support for irrevocable actions (i.e. I/O)
  - Low runtime and space overhead

Introduction

- Treating threads as processes
- Use memory mapped files to share the heap and globals across processes
- Version numbers

Introduction

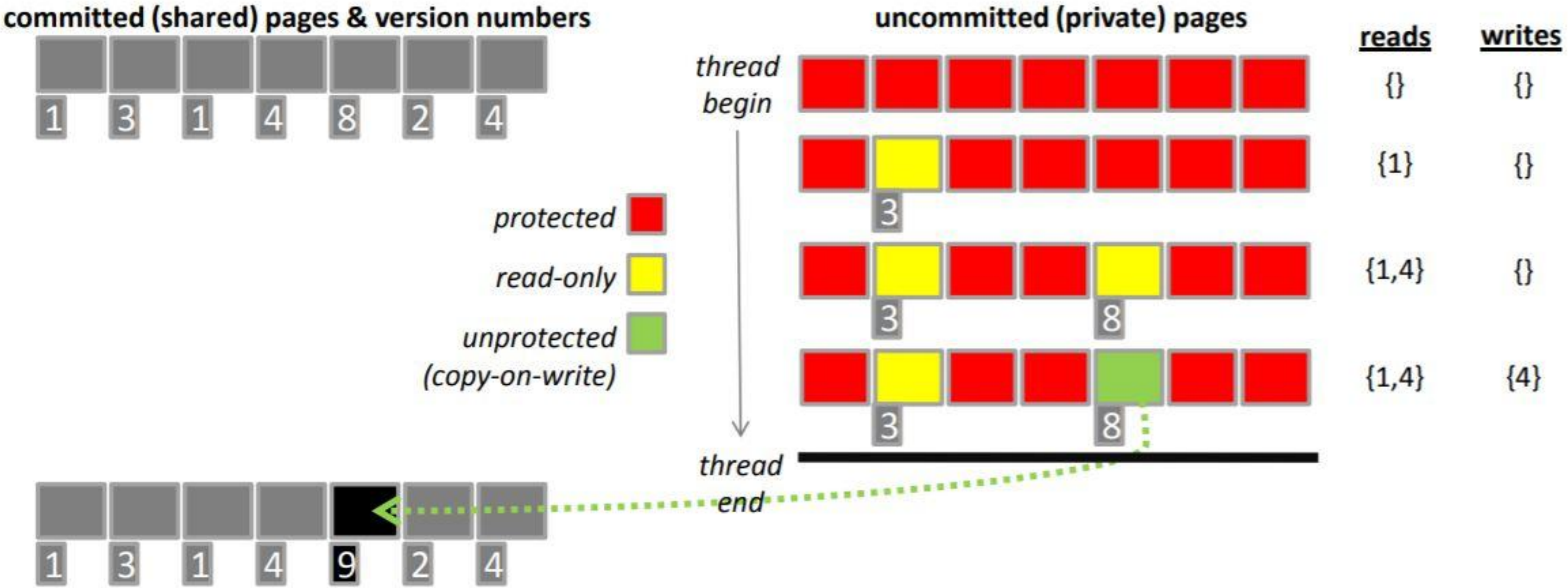
- Globals
- Heap Organization
  - Fixed size heap
  - Sub-heap

Execution -- Initialization

```
void atomicBegin (void) {
  // Roll back to here on abort.
  // Saves PC, registers, stack.
  context.commit();
  // Reset pages seen (for signal handler).
  pages.clear();
  // Reset global and heap protection.
  globals.begin();
  heap.begin();
}
```

Figure 4. Pseudo-code for atomic begin.

Execution



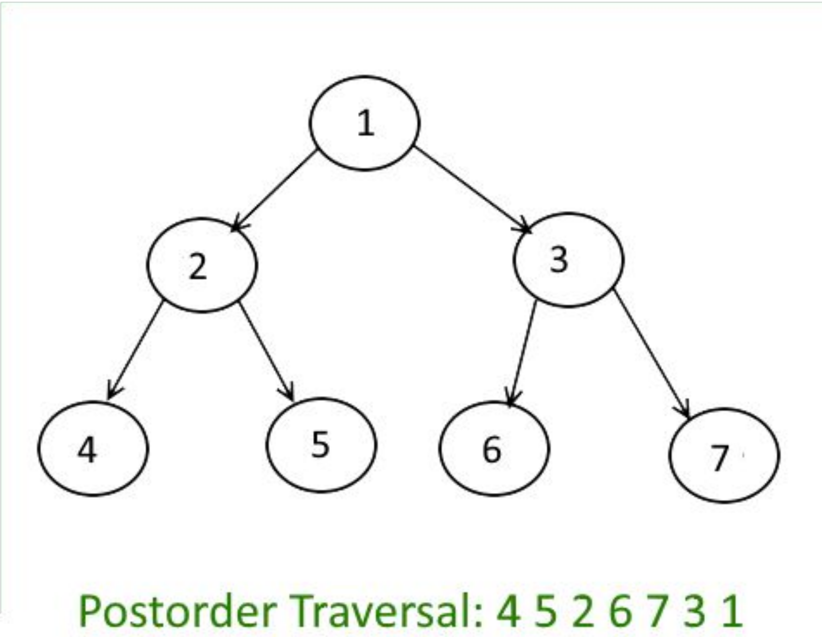
committed (shared) pages & version numbers
1
3
1
4
8
2
4
uncommitted (private) pages
reads
writes
thread begin
{}
{}
3
{1}
{}
protected
read-only
unprotected (copy-on-write)
3
8
{1,4}
{}
3
8
{1,4}
{4}
thread end
1
3
1
4
9
2
4

Execution -- Committing

- Locks are needed（mapping files)
- If version numbers for every page in the read set match the committed versions → Commit
- Else → Rollback

Sequential Commit

- Post-order traversal



Evaluation -- Concurrency Errors

- Deadlocks

```
// Deadlock.
thread1 () {
  lock (A);
  // usleep();
  lock (B);
  // ...do something
  unlock (B);
  unlock (A);
}

thread2 () {
  lock (B);
  // usleep();
  lock (A);
  // ...do something
  unlock (A);
  unlock (B);
}
```

Evaluation -- Concurrency Errors

- Race conditions

```
// Race condition.
int counter = 0;

increment() {
  print (counter);
  int temp = counter;
  temp++;
  // usleep();
  counter = temp;
  print (counter);
}

thread1() { increment(); }
thread2() { increment(); }
}
```

Evaluation -- Concurrency Errors

- Atomicity violations

```
// Atomicity violation.
// thread1
S1: if (thd->proc_info) {
        // usleep();
S2:     fputs (thd->proc_info,..)
    }

// thread2
S3: thd->proc_info = NULL;
```

Evaluation -- Concurrency Errors

- Order violations

```
// Order violation.
char * proc_info;

thread1() {
  // ...
  // usleep();
  proc_info = malloc(256);
}

thread2() {
  // ...
  strcpy(proc_info,"abc");
}

main() {
  spawn thread1();
  spawn thread2();
}
```

```
// Order violation.
int foo;

thread1() {
 foo = 0;
}

main() {
S1:   spawn thread1();
      //  usleep();
S2:   foo = 1;
      // ...
      assert (foo == 0);
}
```

Evaluation -- Real Applications

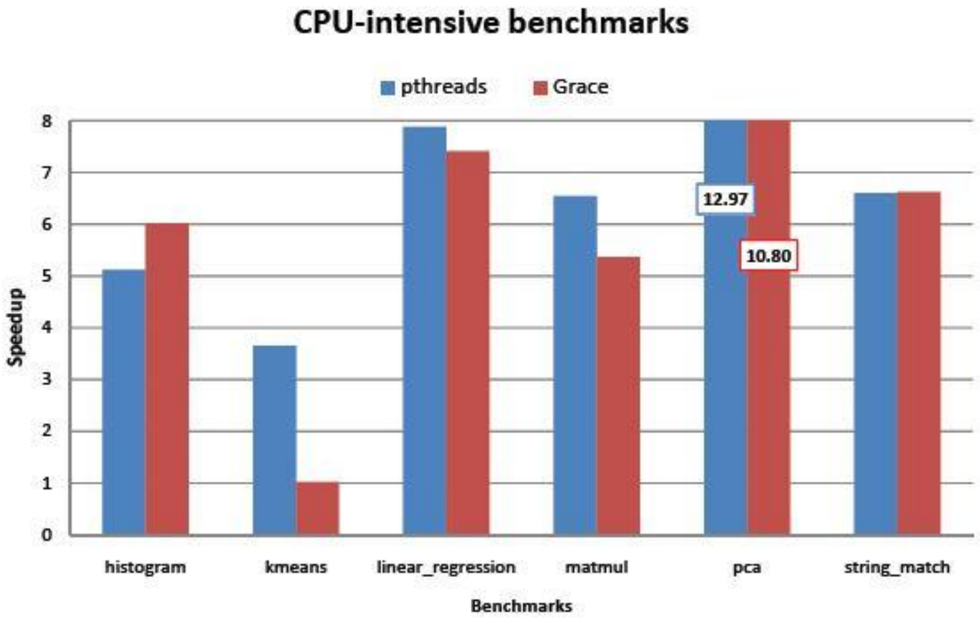
		(average per atomic region)				
Benchmark	**Description**	**Commits**	**Rollbacks**	**Pages Read**	**Pages Written**	**Runtime (ms)**
histogram	Analyzes images' RGB components	9	0	7.3	5.9	1512.3
kmeans	Iterative clustering of 3-D points	6273	4887	404.5	2.3	8.7
linear_regression	Computes best fit line for set of points	9	0	5.6	4.8	1024.0
matmul	Recursive matrix-multiply	11	0	4100	1865	2359.4
pca	Principal component analysis on matrix	22	0	3.1	2.2	0.204
string_match	Searches file for encrypted word	11	0	5.9	4.3	191.1

Table 2. CPU-intensive multithreaded benchmark suite and detailed characteristics (see Section 5.1).

Evaluation -- Real application

- Thread-creation hoisting / argument padding
- Page-size base case
- Changed concurrency structure

Evaluation -- Real application



Evaluation -- Application Characteristics

- Grain size

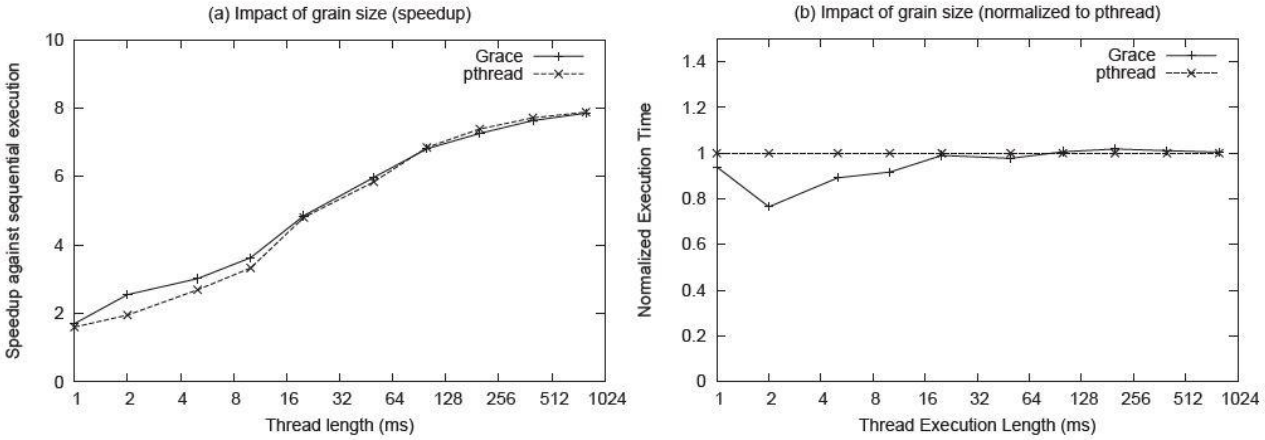


Figure 10. Impact of thread running time on performance: (a) speedup over a sequential version (higher is better), (b) normalized execution time with respect to `pthreads` (lower is better).

Evaluation -- Application Characteristics

- Footprint

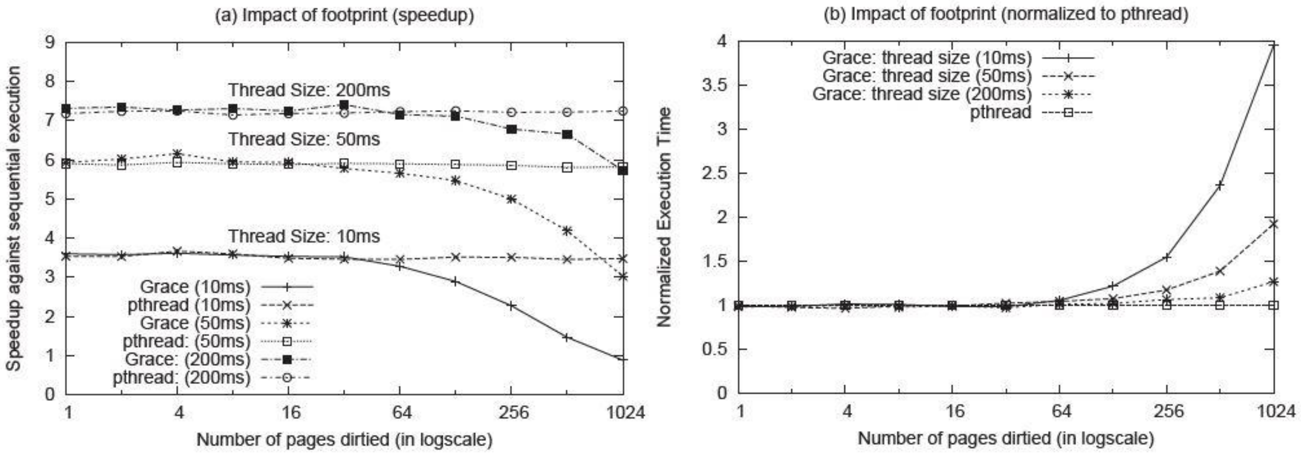


Figure 11. Impact of thread running time on performance: (a) speedup over a sequential version (higher is better), (b) normalized execution time with respect to `pthreads` (lower is better).

Evaluation -- Application Characteristics

- Conflict rate

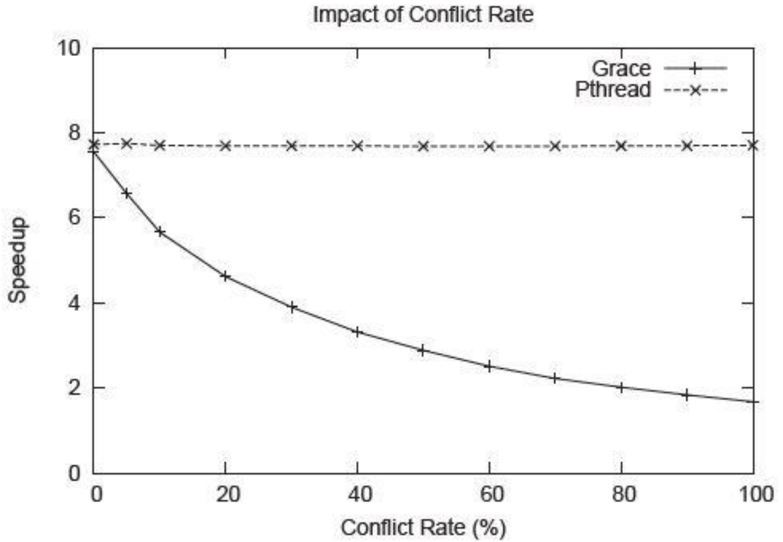


Figure 12. Impact of conflict rate (the likelihood of conflicting updates, which force rollbacks), versus a `pthreads` baseline that never rolls back (higher is better).

Thank you!