Can Transactions Enhance Parallel Programs?

Troy A. Johnson, Sang-Ik Lee, Seung-Jai Min, and Rudolf Eigenmann

Purdue University

Motivation

- Many recent papers on transactions
  - transactions offer a method of parallel programming
- Few comparisons made to languages and compilers for parallel computing
- Similar goal and need for compiler analysis suggest that the underlying technology is closely related

Outline

- Technology Underlying Transactions
- Automatic Program Parallelization
- Comparison
- Improving Automatic Parallelization with Transactions and vice versa
- SPEC OMP with Transactions
- Conclusion

Transactions

- Optimistically-executed atomic block
- Atomic block
  - all effects become visible at the same time
  - not possible to see transient values
- Optimistic execution
  - multiple threads execute the block in parallel
  - some mechanism ensures *atomicity*
    - *may require some threads to re-execute the block if a conflict occurs*

Atomicity

- Not a new concept
  - semaphores, locks, and critical sections are well-known
  - OpenMP has an “atomic” directive
- But language implementations often
  - allow updates to only a single memory cell
  - are pessimistic and use mutual exclusion
- Transactions allow multiple updates and execute optimistically

Performance Benefit

- A transaction allows threads to execute in parallel, even in an atomic block
  - a traditional lock would serialize the threads
- When there are no conflicts, a transaction is faster than a critical section
- When there are many conflicts, the transaction preserves correctness
  - but overhead of repeated re-execution may make it slower than a critical section

Conflict Detection

Static detection

- compiler proves that there will be no conflict and can remove the transaction
- compiler proves that there will be many conflicts and replaces it with a critical section
- compiler fails to prove anything
  - attempts to narrow the conflict set – the set of variables that need to be monitored for conflicts
  - assumes conflicts are infrequent and uses a transaction

Conflict Detection (2)

Dynamic detection

- software or hardware detects conflicts
- live-out thread data is buffered until it can *commit* safely to memory
- size of buffer limits size of transactions
  - execution stalls if the buffer becomes full

Automatic Program Parallelization

- Compile-time parallelization
  - prove independence => use parallel threads
    - same as proving a transaction is unnecessary
  - data-dependence tests used to check for parallel loops
- Compiler-assisted run-time parallelization
  - data-dependence test deferred until run time
  - memory accesses buffered in software
  - code re-executed serially after a conflict

Automatic Program Parallelization (2)

Speculative architectures

- hardware executes speculatively-parallel threads
- data accesses buffered and dependences tracked by hardware
  - buffer commits to memory if speculation succeeds
  - buffer cleared and threads re-execute if a dependence violation (conflict) is detected
- essentially same overheads as transactions
  - conflict detection, re-execution, buffer overflow

Comparison of Transactions and Automatic Parallelization

- Compile-time solutions
  - both use compiler for memory-conflict / data-dependence detection
  - both are most successful for numerical applications and have difficulty with non-numerical programs (e.g., due to pointers, subscripted array indices like A[B[i]])
  - neither have run-time overhead if there are provably no conflicts (dependence violations)
  - both cases exhibit the same strengths and weaknesses

Comparison of Transactions and Automatic Parallelization (2)

- Run-time solutions
  - both schemes rely on compiler's ability to narrow the scope of the data to inspect for conflicts at run-time
  - major challenge for both is to reduce the overhead of run-time conflict detection and re-execution
  - for both, hardware provides conflict detection, temporary buffering, re-execution, and commit

User Model Differences

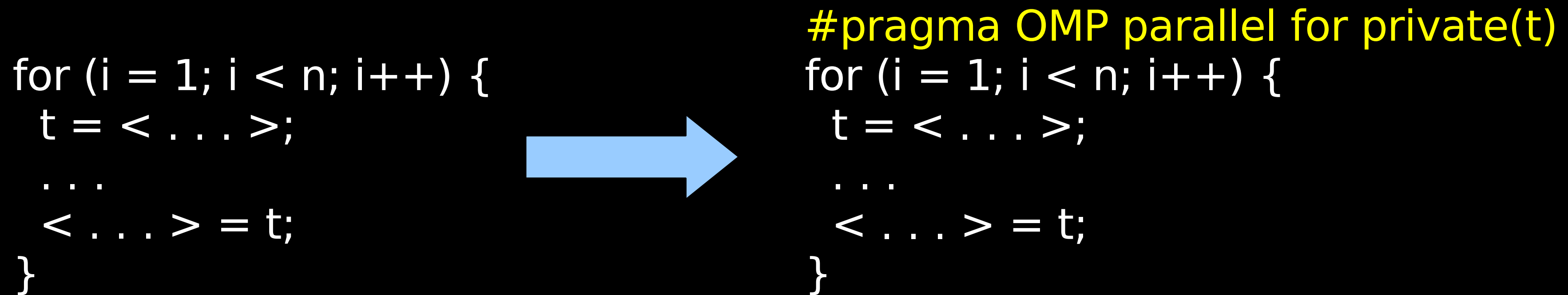
- Transactions
  - programmer inserts into a parallel program
  - focuses the compiler's analysis on certain parts of the program
- Automatic Parallelization
  - begins with a sequential program
  - often needs to analyze the whole program
  - extreme case of whole-program transaction probably requires similar analysis

Improving Parallelization

- Examine three of the most important parallelization techniques
  - data privatization
  - reduction parallelization
  - induction variable substitution
- See how transactions can improve them and vice versa

Data Privatization

Within a loop iteration, variables that are written before they are read can be made private to each thread to remove dependence.



Privatization patterns do not exhibit read-modify-write sequences typical of transactions.

Transactions cannot be used to replace or improve privatization, but privatization can improve transactions by narrowing the conflict set.

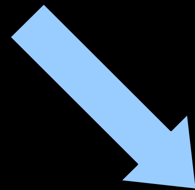
Reduction Parallelization

```
for (i = 1; i < n; i++) {
  sum += < . . . >;
}
```

Conflict every iteration

```
#pragma OMP parallel for
for (i = 1; i < n; i++) {
#pragma OMP atomic
  sum += < . . . >;
}
```

Critical section entered n times



```
#pragma OMP parallel private(lsum)
{
  lsum = 0;
#pragma OMP for
  for (i = 1; i < n; i++) {
    lsum += < . . . >;
  }
#pragma OMP atomic
  sum += lsum;
}
```

Critical section entered once per thread

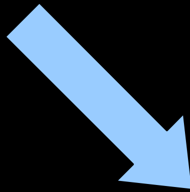
Induction Variable Substitution

```
ind = 5;
for (i = 1; i < n; i++) {
  ind += 2;
  < . . . ind . . . >;
}
```

Conflict every iteration

```
#pragma OMP parallel private (ind)
for (i = 1; i < n; i++) {
  ind = 5 + i * 2;
  < . . . ind . . . >;
}
```

Dependence removed at cost of operation strength increase



```
ind0 = 5;
#pragma OMP parallel private (ind)
  for (i = 1; i < n; i++) {
#pragma OMP atomic
    { ind0 += 2; ind = ind0; }
    < . . . ind . . . >;
  }
```

This transaction implementation avoids strength increase but is possible only if multiple updates can be performed atomically (illegal in OpenMP 2.0)

Transactions with SPEC OMP

- Instrumented sources of mutual exclusion to record number of conflicts
- Conflict Type I
  - omp_set_lock, omp_unset_lock
- Conflict Type II
  - OMP critical, OMP end critical
- Conflict Type III
  - OMP reduction

SPEC OMP Results

Table 1. Conflict analysis of critical sections in SPEC OMP2001. "Conflict prob." expresses the likelihood that a transactional execution will not commit successfully and, instead, roll back and re-execute. "Compile-time dep." indicates whether or not the conflict is certain at compile time.

Id	Benchmark	Source file	TYPE	Conflict prob.	Compile-time dep.
1	ammp	rectmm.c	I	0%	N
2			I	0%	N
3			I	0%	N
4		nonbon.c	I	0%	N
5			I	0%	N
6			I	0%	N
7	gafort	gafort.f90	I	0.02%	N
8			III	23.4%	Y
9	apsi	apsi.f	III	33.0%	Y
10	fma3d	platq.f90	II	8.1%	N
11	wupwise	dznrm2.f	II	33.7%	Y
12		zdotc.f	III	22.9%	Y
13			III	21.8%	Y
14	swim	swim.f	III	25.0%	Y
15	mgrid	mgrid.f	III	27.7%	Y
16	applu	l2norm.f	III	31.2%	Y

SPEC OMP Results (2)

Table 2. When to use Transactions versus Critical Sections: Case 1 is fully parallel. In Case 4, a compiler can detect a dependence. Cases 2 and 3 are the grey area where the compiler can prove neither. The numbers in the Code Examples column refer to Table 1.

Cases	Provably Independent	Provably Dependent	Predicted Conflict	Actual Conflict	Use Trans.?	Use C.S.?	Code Examples
1	T	F	0%	0%	No	No	*
2	F	F		0-1%	Yes	No	1-7
3	F	F		8.1%	No	Yes	10
4	F	T	100%	21.8-33.7%	No	Yes	8-9, 11-16

Transactions vs. Critical Sections

Use transaction when: time[CriticalSection] > time[Transaction]

time[CriticalSection] = NumThreads * time[work]

time[Transaction] = time[work] + MeanReexec * time[work] + time[TOverhe

p	chance of a conflict
$1 – p$	chance of no conflict
$p (1 – p)$	conflict then no conflict
. . .	
$p^k (1 – p)$	k conflicts then no conflict

MeanReexec = Sum{k = 0 to ∞} $k\, p^k (1 – p) = p / (1 - p)$

$$p < \frac{(\text{NumThreads} – 1) * \text{time[work]} - \text{time[TOverhead]}}{\text{NumThreads} * \text{time[work]} - \text{time[TOverhead]}}$$

Conclusion

- Although user models of transactions and automatic parallelization differ, the underlying implementations are the same
- The most important parallelization transformations are also essential for transactions
- We provided guidelines for when to use transactions versus critical sections in SPEC OMP and elsewhere

Can Transactions Enhance Parallel Programs?

Troy A. Johnson, Sang-Ik Lee, Seung-Jai Min,
and Rudolf Eigenman

Purdue University