# Can Transactions Enhance Parallel Programs?*

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

School of Electrical and Computer Engineering
Purdue University, West Lafayette, IN 47907
{troyj, sangik, smin, eigenman}@purdue.edu

**Abstract.** Transactional programming constructs have been proposed as key elements of advanced parallel programming models. Currently, it is not well understood to what extent such constructs enable efficient parallel program implementations and ease parallel programming beyond what is possible with existing techniques. To help answer these questions, we investigate the technology underlying transactions and compare it to existing parallelization techniques. We also consider the most important parallelizing transformation techniques and look for opportunities to further improve them through transactional constructs or – vice versa – to improve transactions with these transformations. Finally, we evaluate the use of transactions in the SPEC OMP benchmarks.

## 1 Transaction-Supported Parallel Programming Models

Although a large number of parallel programming models have been proposed over the last three decades, there are reasons to continue the search for better models. Evidently, the ideal model has not yet been discovered; creating programs for parallel machines is still difficult, error-prone, and costly. Today, the importance of this issue is increasing because all computer chips likely will include parallel processors within a short period of time. In fact, some consider finding better parallel programming models one of today's most important research topics. Models are especially needed for non-numerical applications, which typically are more difficult to parallelize.

### 1.1 Can Transactions Provide New Solutions?

Recently, programming models that include transactional constructs have received significant attention [1, 4, 12, 15]. At a high level, transactions are optimistically executed atomic blocks. The effect of an atomic block on the program state happens *at once*; optimistic execution means that multiple threads can execute the block in parallel, as long as some mechanism ensures atomicity. To this end, both hardware and software solutions have been proposed. An interesting observation is that these contributions make few references to technology

* This work is supported in part by the National Science Foundation under Grants No. 0103582-EIA, and 0429535-CCF.

in languages and compilers for parallel computing. These omissions are puzzling because the two topics pursue the same ultimate goal: making parallel programming easier and more efficient. While the programming models are arguably different, both areas need advanced compiler, run-time, and hardware optimization techniques. Hence, one expects that the underlying techniques supporting these models are closely related. In this paper, we investigate these relationships. We examine how much the concept of transactions can improve parallel program design and implementation beyond existing technology and to what extent transactions are just an interesting new way of looking at the same problem. We also review the ability of existing technology to optimize the implementation of transactions.

### 1.2 The Promise of Programming With Transactions

How can transactional constructs improve parallel programs? A transaction, in its basic meaning, is simply a set of instructions and memory operations. In many situations (e.g., in databases and parallel programming) it is important that the transactions are performed in such a way that their effects become visible simultaneously, or *atomically*. For example, in a bank, it is important that an amount of money gets deducted from one account and put into the other atomically, so that the total balance remains invariant at all times. Similarly, when incrementing a counter by two parallel threads, it is important that reading, modifying, and writing the counter be done atomically.

The concept of atomicity is not new per se. Constructs such as semaphores [5], locks [22], and critical sections [11] have been known for a long time. Nevertheless, language constructs that express atomicity typically allow only single memory updates (e.g., the OpenMP [21] `atomic` directive). Blocks of atomic memory operations are expressed through critical sections, which prevent concurrent execution of the block. This implementation is conservative or "pessimistic." The new promise of transactions is to eliminate some of the disadvantages that come with state-of-the-art constructs, namely reducing overhead through "optimistic execution" (if threads end up not accessing the same data inside a critical section, they should execute concurrently) and managing locks (avoiding deadlock and bookkeeping of multiple locks). These overheads naturally occur, as programs are written conservatively. For example, a banking software engineer may protect all account operations with one critical section, even though it could be known, in some cases, that the operations happen to different classes of accounts. The engineer may optimize the accounting software by creating separate locks for the two account classes; however, this increases the amount of bookkeeping information and requires more effort to avoid deadlocks.

The new idea behind transactions is that the programmer can rely on an efficient execution mechanism that executes in parallel *whenever possible*. Thus, the programmer uses the same "critical section" everywhere by simply writing *atomic*. At run time, two account operations or loop-counter updates can occur simultaneously. If different accounts are accessed or different counters are updated, then the program continues normally; if the same account or same

counter is updated, then the transaction's implementation properly orders the operations. It is the transaction implementation's responsibility to provide efficient mechanisms for detecting when concurrency is possible and for serializing the operations when necessary.

Two questions arise: (i) Are transactions an adequate user model, and (ii) can transactions be implemented efficiently? Although the idea of an atomic language construct is not new [20], only time and experience can answer whether programmers find transactions useful. Today, only few real programs have been written with transactional constructs. An important challenge is that much parallel programming experience exists in the area of numerical programs; however, transactions aim at all classes of programs. The second question is the focus of this paper. Our thesis is that the technology underlying efficient transactions is very similar to the one that exists today for program parallelization – parallelizing compiler techniques [3, 9], implementation techniques of parallel language constructs [18], and hardware techniques for speculative parallelization [8, 10]. The ultimate question for the language and compiler community is whether or not we have missed something that we can now learn from the ideas behind transactional constructs. If so, we may be able to incorporate that new knowledge into our compilers, run-time systems, and supporting hardware.

## 2 Comparing the Technology Underlying Transactions and Program Parallelization

### 2.1 Technology Underlying Transactions

Within transactions, threads that do not conflict should execute in parallel unhindered. Conflict detection is therefore at the heart of implementation technology for transactions. Conflict detection can be performed statically or dynamically.

Static conflict detection relies on the compiler's ability to tell that threads access disjoint data. Provably non-conflicting threads can execute safely in parallel without the guard of a transaction; the compiler can remove the transaction altogether. The compiler also may remove conflict-free code out of the transaction, hence narrowing the guarded section. This optimization capability is important because it allows the programmer to insert transactions at a relatively coarse level and rely on the compiler's ability to narrow them to the smallest possible width. Furthermore, if a compiler can identify instructions that always conflict, it may guard these sections directly with a classical critical section. Applying common data dependence tests for conflict resolution is not straightforward, as conflicts among *all* transactions must be considered. For *strong atomicity* [4] this analysis is even necessary between transactions and all other program sections. Note that common data-dependence tests attempt to prove independence, not dependence; i.e., failure to prove independence does not imply dependence.

Compile-time solutions are highly efficient because they avoid run-time overhead. Nevertheless, their applicability is confined to the range of compile-time

analyzable programs. Often, these are programs that manipulate large, regular data sets – typically found in numerical applications. Compile-time conflict resolution is difficult in programs that use pointers to manipulate dynamic data structures, which is the case for a large number of non-numerical programs.

For threads that are not provably conflict-free, the compiler still can assist by narrowing the set of addresses that may conflict. At run time, this conflict set must be monitored. The monitoring can happen either through compiler-inserted code (e.g., code that logs every reference) or through interpreters (e.g., virtual machines). At the end of the transaction, the logs are inspected for possible conflicts; in the event of a conflict, the transaction is rolled back and re-executed. Rollback must undo all modifications and can be accomplished by redirecting all write references to a temporary buffer during the transaction. The buffer is discarded upon a rollback; a successful transaction commits the buffer to the real address space. Again, interpreters may perform this redirection of addresses and the final commit operation on-the-fly. Evidently, there is significant overhead associated with software implementations of transactions, giving rise to optimization techniques [1, 12].

Fully dynamic implementations of transactions perform conflict detection, rollback and commit in hardware. During the execution of a transaction, data references are redirected to a temporary buffer and monitored for conflicts with other threads' buffers. Detected conflicts cause a rollback, whereby the buffer is emptied and threads are restarted. At the end of a successful, conflict-free transaction, the thread's buffer is committed. Conflict detection in hardware is substantially faster than software solutions, but still adds extra cycles to every data reference. The cost of a rollback is primarily in the wasted work attempting the transaction. Commit operations may be expensive, if they immediately copy the buffered data (for speculative parallelization, hardware schemes have been proposed to commit in a non-blocking style, without immediate copy [24]). An important source of overhead stems from the size of the buffer. While small hardware buffers enable fast conflict detection, they may severely limit the size of a transaction that can be executed. If the buffer fills up during a transaction, parallel execution stalls.

### 2.2 Technology Underlying Program Parallelization

A serial program region can be executed in parallel if it can be divided into multiple threads that access disjoint data elements. Implementing this concept requires techniques analogous to the ones in Section 2.1. There are compile-time, compiler-assisted run-time, and hardware solutions.

*Compile-time parallelization:* Data-dependence analysis is at the heart of compile-time, automatic parallelization. Provably independent program sections can be executed as fully parallel threads. The analysis is the same as what is needed for conflict detection of transactions. Data-dependence tests have proven most successful in regular, numerical applications; data dependence analysis in the presence of pointers [13] is still a largely unsolved problem. Where successful,

automatic parallelization is highly efficient, as it produces fully-parallel sections, avoiding run-time overheads.

*Run-time data-dependence tests:* These tests [23] have been introduced to defer the detection of parallelism from compile time to run time, where the actual data values and memory locations are known. Run-time data-dependence tests select the arrays to be monitored and insert monitoring code at compile time. At run time, memory references are recorded; if a conflict is detected, the parallel section is rolled back, usually followed by a serial execution. Run-time data dependence tests are efficient when the address space to be monitored is small. As this is not often the case, these methods are difficult to apply in general.

Hardware parallelization mechanisms are also known as speculative architectures [7, 16]. They execute potentially independent threads in parallel, while tracking conflicts. Upon a conflict the thread is rolled back. During the speculative execution, memory references are redirected to a speculation buffer [8], which is committed to the actual memory upon successful speculation (no conflicts detected) or cleared (upon rollback). Hardware speculation mechanisms have essentially the same overheads as mentioned above for hardware transactions: data dependence tracking adds a cost to each memory reference, rollbacks represent significant overheads, and speculation buffer overflow is a known problem.

### 2.3 Comparison

*Compile-time solutions* Transactions and automatic program parallelization models need implementation technologies that are very similar. Compile-time solutions hinge on the compiler's ability to detect memory conflicts – or data dependences. It can be expected that, for both models, this solution succeeds in regular, numerical programs, whereas pointer-based, non-numerical code patterns pose significant challenges. In terms of efficiency, neither transactions nor automatic parallelization seem to offer advantages over the other model. For both models, static, compile-time solutions – where applicable – are most efficient, as they are void of run-time overheads. They also exhibit the same weaknesses in irregular and pointer-based programs.

*Run-time solutions* Compiler-assisted run-time solutions underlie both software-based transaction schemes and run-time parallelization techniques. Both schemes rely on the compiler's ability to narrow the range of data that needs to be inspected at run time. Many techniques have been proposed to perform the inspection; the big challenge is to reduce run-time overhead. Sophisticated bit-manipulating inspection code has been used in run-time data-dependence tests [23]. Other techniques detect if re-invocations of the same code regions happen under the same data context, in which case the serial or parallel outcome is already known and reinspection becomes unnecessary [17]. These techniques are compiler-based. Interpreters and virtual machines are most flexible in performing inspection and conflict analysis; however, their performance has yet to be proven.

Hardware-assisted schemes must provide the same basic mechanisms for transactions and speculative parallelization: data-dependence tracking, temporary buffering, rollback, and commit. The associated overheads are essentially the same. Adaptive synchronization techniques have been proposed for speculative synchronization [19], as effective means to eliminate repeated rollback. The same mechanisms would be effective for transactions. A subtle difference stems from the order of thread execution. The effect of a speculatively parallelized program must be the same as in its serial execution. This requirement is most easily implemented by committing the threads in the order that they would execute in the serial program version. By contrast, as transactions are entered from within a parallel program, correctness demands no such order. This might allow for a more efficient implementation, as we will discuss further in Section 3.

*Differences stemming from the user models* While the underlying technology is very similar, interesting differences lie in the user models. Transactions are embedded inside a program that is already parallel. By contrast, automatic parallelization and speculative parallelization start from a sequential program; the compiler has the additional task of partitioning the program into potentially parallel threads. Parallelizing compilers commonly perform this task at the level of loops, considering each loop iteration as a potential parallel thread. Partitioning techniques for speculative parallelization have been developed that split programs so as to maximize parallelism and minimize overheads [14, 26]. Another difference resulting from the user model is that, by explicitly parallelizing a program and inserting transactional regions, the programmer focuses the compiler and run-time system's attention on specific code sections, whereas automatic or implicit parallelization must analyze the entire program. The tradeoffs between automatic and manual parallelization are well-known. Automatic parallelization has been most successful in regular, numerical programs, and similarly for speculative parallelization. As transactional models aim at a broad class of programs, explicit parallelization may be a necessity. Evidently, there is an important tradeoff: large transactions are user-friendly, but lose the important advantage of focusing the compiler's attention on small regions. The extreme case of a whole-program transaction likely requires compiler optimizations similar to those for automatically parallelizing whole programs. It is also worth noting that the most advanced optimization techniques require whole-program analysis, even if their goal is to improve a small code section. For example, interprocedural pointer analysis [27] may gather information from other subroutines that helps improve a small transaction. Hence, once we develop highly-optimized techniques, implementing the different user models becomes increasingly similar.

## 3 Improving Parallelization Techniques through Transactions and Vice Versa

Three parallelization techniques have proven most important [6]: data privatization, parallelization of reduction operations, and substitution of induction

variables. This section discusses opportunities for improving these techniques through the use of transactions and vice versa.

### 3.1 Data Privatization

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

**Fig. 1.** Simple form of a program pattern that is amenable to privatization and its parallel form, expressed in the OpenMP directive language: To perform this transformation, the compiler or programmer must recognize that `t` is *defined* before it is *used* in every loop iteration. No true dependence exists across loop iterations. A more complex form of privatizable data would have `t` as an array; the compiler would have to analyze the subscripts of the references defining and using `t`.

Privatization [25] is the most widely applicable parallelization technique. It recognizes data values that are only used temporarily within a parallel thread and thus are guaranteed not to be involved in true data dependences across threads. In data dependence terms, the technique removes anti-dependences, which occur because two or more threads use the same storage cell to hold different values. The privatization technique creates a separate storage cell for each thread (through renaming or `private` language constructs), thus eliminating the storage-related dependence. Figure 1 shows a program pattern that is amenable to privatization. Such patterns do not exhibit read, modify, and write sequences typical of transactions. Transaction concepts cannot be used to improve or replace privatization.

By contrast, privatization is important for optimizing transaction implementations. Variables that can be recognized as private can be removed from the conflict set. They do not need to be monitored for conflicts and their accesses never necessitate a rollback; however, private data still needs to be redirected to the temporary storage during the execution of the transaction. If the data is not used after the transaction (i.e, not live-out of the transactional region), it also does not need to be committed. Notice that live-out private data leads to a race condition (by program semantics), as the value of the transaction that happens to complete last will prevail.

The lack of a program order again differentiates the implementation of a transaction from parallelizing a sequential region in subtle ways. (i) In a parallelized program, the compiler and run-time system must ensure that the value of the *youngest* thread prevails. (ii) In speculative parallelization, private data – even if it is not recognized as such – never necessitates a rollback. Only a

write reference following a premature read reference is a conflict (thus, anti-dependence violations are not a problem). The presence of a (sequential) program order clearly defines what is premature. Anti-dependences are implicitly enforced through the speculative buffering mechanism and the commit actions, which happen in program order. For transaction implementations, the absence of a program order dictates that all read and write references to the same address cause rollbacks. Privatization is essential to eliminate such overheads.

### 3.2 Reduction Parallelization

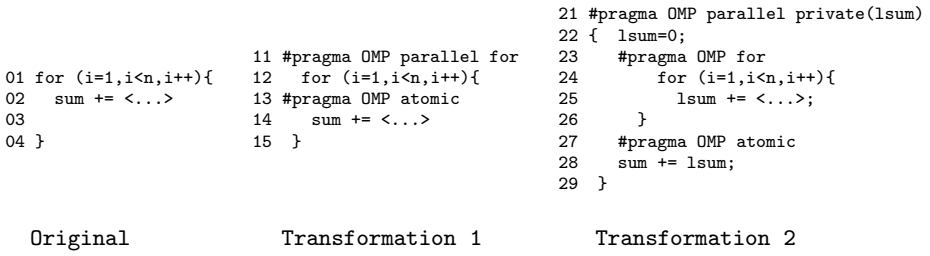


**Fig. 2.** Reduction pattern and parallel form expressed in OpenMP: notice that according to OpenMP semantics, statements 22 and 28 are executed once per processor, whereas the processors share the iterations of loops 12 and 24.

Figure 2 shows a reduction program pattern and two forms of parallel transformations. While Transformation 1 looks more elegant, the absence of efficient implementations of the `atomic` construct (often a software implementation of a critical section) make Transformation 2 the preferred option. In the latter form, the atomic section is entered once per processor versus $n$ times in Transformation 1. The size of the reduction, $n$, generally must be large for the transformation to be beneficial.

Consider a transactional implementation of the atomic construct: in Transformation 1, the transaction will never proceed in parallel, as there is always a conflict on `sum`. There will be only some parallelism, if the processors happen to enter the transaction at different times, due to load imbalance. The same holds when using a critical section. Furthermore, the overheads of transactions (conflict tracking and rollbacks) make a plain critical section the much preferred choice.

If the expression `<...>` involves a substantial computation (e.g., a function call), its concurrent execution may be beneficial. To exploit this parallelism, the programmer or compiler moves the expression out of the transaction into the fully-parallel part of the code. Transformation 2 achieves exactly this effect. Hence, this transformation also yields an optimized form of a reduction implemented by a transaction.

For array (or irregular) reductions, the situation is different. The variable `sum` in Figure 2 would have a subscript, such as `sum[expr]`, where `expr` is a

loop-variant expression. In this case, different loop iterations modify different elements of `sum`, hence a transactional implementation of the atomic construct in Transformation 1 can exploit some parallelism. The sparser the reduction (i.e., `expr` is different in more iterations), the more parallelism is exploited. Transformation 2 (not shown for array reductions, but similar to Figure 2) will incur substantial overhead, as the variable `lsum`, which is now also an array, may be large, making the additional statements 22 and 28 expensive. Hence, Transformation 1 with transactions would be preferred for sparse array reductions, whereas the classical transformation with a plain critical section is preferred for dense array reductions. Deciding this tradeoff is difficult, as the degree of sparsity is not likely to be known at compile time.

### 3.3 Induction Variable Substitution

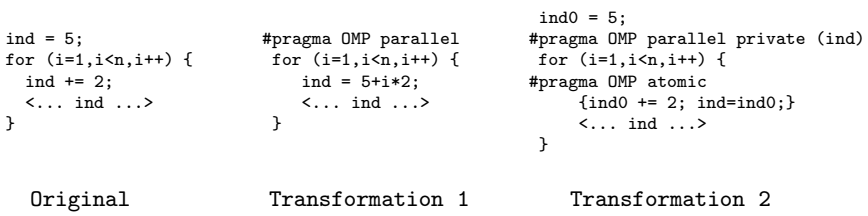


**Fig. 3.** Induction Variable Substitution: Transformation 1 is the common parallelizing transformation. Transformation 2 is possible, if the loop variable $i$ is not used in the loop body. Notice that the atomic block requires two memory cells to be updated, which is not currently supported by OpenMP.

Induction variable substitution removes data dependences at the cost of increasing the strength of the computation. In Figure 3, the original loop has cross-iteration data dependences. These dependences are removed in the transformed version, but the expression $5 + i * 2$ now involves a multiplication instead of just an addition. To obtain parallelism without increasing strength, one might consider guarding the original induction statement with a transaction, capturing the resulting value in a private variable, as in Transformation 2. This code version exploits parallelism among the statements `<... ind ...>`, which may be beneficial if this computation is large compared to the induction statement. Nevertheless, the same parallelism also would be exploited with an implementation of atomic through a plain critical section. In fact, this version would be more efficient; whenever two threads enter the transaction, there is a conflict, making a critical section the best implementation.

## 4 Evaluating Transactions for the SPEC OMP Benchmarks

SPEC OMP2001 [2] is a benchmark suite used for the performance evaluation and comparison of Shared-Memory Multiprocessor(SMP) systems and consists

```
!$OMP PARALLEL PRIVATE(rand, i, tmp1, tmp2)
!$OMP DO
  DO j=1,npopsiz-1
     CALL ran3(rand)
     iother=j+1+DINT(DBLE(npopsiz-j)*rand)
     IF (j < iother) THEN
        CALL omp_set_lock(lck(j))
        CALL omp_set_lock(lck(iother))
     ELSE
        CALL omp_set_lock(lck(iother))
        CALL omp_set_lock(lck(j))
     END IF
     tmp1 = iparent(iother)
     iparent(iother) = iparent(j)
     iparent(j) = tmp1
     tmp2=fitness(iother)
     fitness(iother)=fitness(j)
     fitness(j)=tmp2
     IF (j < iother) THEN
        CALL omp_unset_lock(lck(iother))
        CALL omp_unset_lock(lck(j))
     ELSE
        CALL omp_unset_lock(lck(j))
        CALL omp_unset_lock(lck(iother))
     END IF
  END DO
!$OMP END PARALLEL DO
```

(a) TYPE I - "gafort.f90" from gafort

```
!$OMP CRITICAL
  IF (SCALE .LT. LSCALE) THEN
     SSQ = ((SCALE/LSCALE)**2)*SSQ+LSSQ
     SCALE = LSCALE
  ELSE
     SSQ = SSQ+((LSCALE/SCALE)**2)*LSSQ
  END IF
!$OMP END CRITICAL
```

(b) TYPE II - "dznrm2.f" from wupwise

```
  EXNER = 0.0
!$OMP PARALLEL PRIVATE(J,K,T_EXNER)
  T_EXNER = 0.0
!$OMP DO
  DO J=1, NY
     DO K=2, NZ
        ..
        T_EXNER = T_EXNER + P(I,J,...)
        ..
     ENDDO
  ENDDO
!$OMP END DO
!$OMP ATOMIC
  EXNER = EXNER + T_EXNER
!$OMP END PARALLEL
```

(c) TYPE III - "apsi.f" from apsi

**Fig. 4.** Critical section types

of three C applications and nine FORTRAN applications, including *gafort*, a non-numerical application. Each of the applications in SPEC OMP2001 is either automatically or manually parallelized using OpenMP directives. There are three types of critical sections in SPEC OMP2001, which have been implemented using lock-based synchronization. Figure 4 illustrates examples for each type of critical section. Type I is the critical section guarded by the omp_set_lock() and omp_unset_lock() OpenMP API runtime library routines and type II is the critical section specified by the *OMP CRITICAL* and *OMP END CRITICAL* directives. Both type I and type II exist in source form in SPEC OMP2001, whereas type III is the critical section generated by the underlying OpenMP implementation when converting *OMP reduction* into an efficient form as depicted in Figure 2. While these critical sections can be converted into transactions with little effort, the runtime overheads may offset the benefit of the transaction. In order to estimate these overheads, we measured the probability $p$ that a transaction would conflict and thus roll back. To this end, we count the number of runtime conflicts that happened in the current critical section implementation. We performed our experiments on a four-processor 480MHz SPARC machine using the *ref* dataset of the benchmarks.

Type I critical sections are found six times in *ammp* and once in *gafort*. In *ammp*, Type I critical sections access the same shared data structure type through pointers, which are dynamically changing at runtime. From the program code, one cannot determine whether or not conflicts will arise. In *gafort*,

**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 |

as shown in Figure 4 (a), a critical section guards the swap of an array element with a randomly chosen element; due to the random choice, conflicts cannot be determined statically. To determine the conflict probability, we created a wrapper function of the *omp_set_lock()* and *omp_unset_lock()* library routines, respectively. When there is a thread $T$ trying to enter a critical section, the wrapper function of *omp_set_lock()* function is called and it checks if there are other threads executing within a critical section, accessing the same shared memory location as the thread $T$ is going to access. From our experiment, we found that there are no conflicts in all six critical sections in *ammp*. The critical section in *gafort* has a very small number of conflicts, less than 1% of the total number of invocations of the critical section. Nevertheless, the critical section requires a very large array of locks, the `lck` array in Figure 4 (a), for correct execution.

Type II critical sections are found in *wupwise* and *fma3d*. The conflict probability of the critical section from *wupwise*, shown in Figure 4 (b), is only 33.7%, even though the critical section contains a statically known dependence on the shared variable *SSQ*. Due to load imbalance, the parallel threads enter the critical section at different times. The *fma3d* code has a large critical section that contains a small loop. Although it cannot be proven at compile-time, there is a dependency to the shared array, *MATERIAL(*)*, in the critical section. Despite this dependency, the critical section in *fma3d* exhibits a low conflict probability of 8.1% that we also attribute to load imbalance.

We transformed all *OMP reduction* clauses found in SPEC OMP2001 into the *Transformation 2* form of Figure 2 so that the transformed code consists of a parallel loop and a critical section pair. These Type III critical sections contain reduction statements, where a reduction variable is a source of conflicts from a transaction point of view. The conflict probability of Type III critical sections varies between 21.8% and 33% depending on the applications. Again, load imbalance frequently prevents the threads from entering the section simultaneously.

**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 |

One of the key assumptions of transactional memory is that most transactions commit successfully. Table 1 shows that, for Type I critical sections, the probability of successful commit is almost 100%, which coincides with the cases where compile-time analysis cannot prove the absence of conflicts. Hence, these critical sections are good candidates for transactions. In most Type II and Type III cases, the existence of dependences is provable at compile time, except in *fma3d*. When there is a dependency in the critical section, the probability of conflict ranges from 8.1% to 33.7%.

Table 2 summarizes our findings. The four cases differ by the available compile-time knowledge. Case 1 is statically known to be non-conflicting. The compiler can remove all synchronization. Case 4 has a compile-time provable dependence. Critical sections are always the preferred choice. Recall that, due to load imbalance, runtime conflicts happen only 21.8-33.7% of the time. Case 2 has statically unknown dependences. Transactions are beneficial and conflict with a small likelihood of 0-1%. Case 3 has a dependence, but it is not detectable with state-of-the-art compiler technology. A transactional implementation would conflict 8.1% of the time; again, load imbalance prevents many conflicts. In actuality, given the dependence, a critical section implementation would be better.

The above guidelines for when to use a transaction versus a critical section are very loose; the actual tradeoff depends on the efficiency of the transaction and critical section implementations. The tradeoff that must be made is:

$$time[CriticalSection] > time[Transaction]$$

When the inequality is true, a transaction should be used. At a greater level of detail, the inequality becomes:

$$N{*}time[work] > time[TSetup]{+}time[work]{+}MeanReexec{*}time[work]{+}time[TCommit]$$

On the left side of the inequality above, $N$ is the number of threads and *work* represents the code within the critical section. The critical section serializes the *work*, so the total execution time is $N * time[work]$. On the right side of the inequality above, $TSetup$ is any setup work that needs to be done to initiate the transaction, *work* represents the code within the transaction, $MeanReexec$ is the average number of times the transaction will be re-executed, and $TCommit$ is any work that needs to be done to complete the successful transaction. Ideally, there are no conflicts during the transaction and, assuming perfect overlap, the execution time for the work is $time[work]$. For each re-execution, there is an additional $time[work]$.

Because the *time* function yields values that are either implementation or application-specific, the inequality cannot be made much more detailed, but we can estimate the $MeanReexec$ given the conflict probability $p$. Due to fewer threads re-executing as some manage to successfully commit, $p$ in fact may not be constant, but we approximate it as the probability of a conflict during the transaction's first attempt. Thus, the chance of no conflict is $1 - p$, the chance of one conflict followed by no conflict is $p(1 - p)$, and so forth such that the chance of $k$ conflicts is approximated as $p^k(1 - p)$. $MeanReexec$ is found using the weighted infinite sum $\sum_{k=0}^{\infty} kp^k(1 - p) = \frac{p}{1-p}$. Therefore, the inequality becomes:

$$N{*}time[work] > time[TSetup]{+}time[work]{+}\frac{p}{1-p}{*}time[work]{+}time[TCommit]$$

Observe that for $p = 0$, the inequality compares an ideal execution of a transaction (i.e., no re-executions) to a critical section. For that case, unless $time[work]$ is very brief or the transaction's implementation is inefficient, it is best to use a transaction. For some greater value of $p$, it becomes better to use a critical section. Solving for $p$ and then simplifying, we obtain the break-even point:

$$p < \frac{(N-1) * time[work] - time[TSetup] - time[TCommit]}{N * time[work] - time[TSetup] - time[TCommit]}$$

When this inequality is true, it is more efficient to use a transaction. The inequality has several interesting properties. The limit of the fraction is 1 as $N \rightarrow \infty$, so for a very large number of processors, it nearly always will be better to use a transaction. As transactional overhead becomes very small ($time[TSetup] \rightarrow 0$ and $time[TCommit] \rightarrow 0$), or as the amount of work becomes very large ($time[work] \rightarrow \infty$), the fraction becomes $\frac{N-1}{N}$. For example, for $N = 4$, $p < 0.75$ implies that a transaction should be used. Programs can be optimized using this inequality by obtaining $time[work]$ and $p$ from a profile run of the application. The $time[TSetup]$ and $time[TCommit]$ values can be obtained a single time (i.e., they are application-independent) by profiling an empty transaction.

## 5 Conclusions

We have compared the technology underlying transactions and program parallelization. We find that, while the two user models differ, the underlying implementation technology is essentially the same. Advanced optimizations are necessary for both models. We reviewed the most important optimization techniques for parallel programs and found that these techniques are essential for optimizing transaction implementations as well.

We have analyzed the SPEC OMP benchmarks for the applicability of transactions. While many locks and critical sections could be replaced by transactions, this replacement is beneficial in only a few cases – where data dependences are statically unknown. In provably-independent code sections, synchronization can be eliminated; in provably-dependent code, critical sections are preferred over transactions. We have also found many cases with a definite conflict, but a low runtime conflict probability, suggesting that transactions may be beneficial. In actuality, it is load imbalance that prevents the section from simultaneous access; critical sections are the preferred choice.

Our evaluation has focused on the SPEC OMP benchmarks, which contain numerical applications, in all but one code. Compiler optimizations for parallel programs are most mature in this application area. Non-numerical programs are known to pose substantial challenges for compiler optimizations. As implementation technology for transactions is very similar, we expect this challenge to hold. Studies similar to the one presented in this paper are needed for this large and growing area of parallel programs.

## References


1. A.-R. Adl-Tabatabai et al. Compiler and Runtime Support for Efficient Software Transactional Memory. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 26–37, June 2006.
2. V. Aslot et al. SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. In *Proc. of the Workshop on OpenMP Applications and Tools (WOMPAT2001), Lecture Notes in Computer Science, 2104*, pages 1–10, July 2001.
3. W. Blume et al. Parallel Programming with Polaris. *IEEE Computer*, pages 78–82, December 1996.
4. B. D. Carlstrom et al. The ATOMOS Transactional Programming Language. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 1–13, June 2006.
5. E. W. Dijkstra. Co-operating Sequential Processes. In F. Genuys, editor, *Programming Languages*, pages 43–112, 1968.
6. R. Eigenmann, J. Hoeflinger, and D. Padua. On the Automatic Parallelization of the Perfect Benchmarks. *IEEE Transactions of Parallel and Distributed Systems*, 9(1):5–23, Jan. 1998.
7. M. Franklin. *The Multiscalar Architecture*. PhD thesis, University of Wisconsin-Madison, November 1993.
8. S. Gopal et al. Speculative Versioning Cache. In *4th IEEE Symposium on HPCA*, pages 195–205, February 1998.

9. M. W. Hall et al. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, pages 84–89, December 1996.
10. L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *Proc. of the 8th International Conference on ASPLOS*, 1998.
11. P. B. Hansen. *Operating System Principles*. Prentice Hall, 1973.
12. T. Harris et al. Optimizing Memory Transactions. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 14–25, June 2006.
13. J. Hummel, L. J. Hendren, and A. Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 218–229, New York, NY, USA, 1994. ACM Press.
14. T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar. Min-Cut Program Decomposition for Thread-Level Speculation. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 59–70, June 2004.
15. S. Kumar et al. Hybrid Transactional Memory. In *Proceedings of the Symposium on Principles and Practices of Parallel Programming*, pages 209–220, 2006.
16. Lance Hammond and Basem A. Nayfeh and Kunle Olukotun. A Single-Chip Multiprocessor. *IEEE Computer*, 30(9):79–85, 1997.
17. S.-J. Min and R. Eigenmann. Combined Compile-time and Runtime-driven Proactive Data Movement in Software DSM Systems. In *Proc. of Seventh Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers (LCR2004)*, pages 1–6, 2004.
18. S.-J. Min, S. W. Kim, M. Voss, S.-I. Lee, and R. Eigenmann. Portable Compilers for OpenMP. In *OpenMP Shared-Memory Parallel Programming*, Lecture Notes in Computer Science #2104, pages 11–19, Springer Verlag, July 2001.
19. A. Moshovos et al. Dynamic Speculation and Synchronization of Data Dependences. In *Proceedings of the 24th ISCA*, pages 181–193, June 1997.
20. OpenMP Architecture Review Board. OpenMP Fortran Application Program Interface, October 1997.
21. OpenMP Architecture Review Board. OpenMP Application Program Interface. http://www.openmp.org/, May 2005.
22. G. L. Peterson. Myths about the Mutual Exclusion Problem. *Information Processing Letters*, 12(3):115–116, June 1981.
23. L. Rauchwerger and D. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *The ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI'95)*, pages 218–232, June 1995.
24. G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *The 22th International Symposium on Computer Architecture (ISCA-22)*, pages 414–425, June 1995.
25. P. Tu and D. Padua. Automatic Array Privatization. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Proc. Sixth Workshop on Languages and Compilers for Parallel Computing, Portland, OR. Lecture Notes in Computer Science.*, volume 768, pages 500–521, August 12-14, 1993.
26. T. N. Vijaykumar and G. S. Sohi. Task Selection for a Multiscalar Processor. In *Proc. of the 31st International Symposium on Microarchitecture*, December 1998.
27. J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 131–144, New York, NY, USA, 2004. ACM Press.