Source Code Transformations to Increase the Performance of Software Transactional Memory

Stefan Kempf, Ronald Veldema, and Michael Philippsen

University of Erlangen-Nuremberg, Computer Science Dept., Programming Systems Group, Martensstr. 3 • 91058 Erlangen • Germany
{stefan.kempf, veldema, philippsen}@cs.fau.de

Software Transactional Memory (STM) promises to simplify concurrent programming. But its use incurs a number of overheads compared to lock based strategies: the cost of recording writes in a transactional log and the cost of a rollback in case of a failing commit. Instead of improving the STM itself, our orthogonal approach uses source code transformations on atomic blocks that let the code make better use of the underlying STM. We examine a number of source code transformations to reduce both sources of overheads. Because the optimizations work at source-level, they are portable.

We use a simplified C++ dialect (TransC) and target an STM implementation similar to TL2 [1]. TransC supports atomic blocks, whose effects on memory appear to happen atomically. Our STM records all memory accesses to a log. In the commit phase (the end of the atomic block) TransC acquires the locks for all involved addresses before it applies the writes from the log. If the locks cannot be acquired, the transaction fails.

Without the source code transformations, our STM achieves comparable performance to TL2 and SwissTM [2] over a set of benchmarks, including those from [3].

Our six novel source code transformations greatly reduce STM overheads (up to 68%) and can be applied to other STMs as well. The selection of which transformations to apply requires program knowledge and is best left to the programmer.

Six novel Source Code Transformations

1. Piggybacking statements of atomic blocks onto the commit: all statements at the end of an atomic block with compile-time fixed addresses are moved to a separate function that is executed within the commit phase, with the necessary locks pre-acquired. If the statements form a contentious part of a transaction, this transformation is beneficial because the moved statements do not cause early aborts any longer. This transformation saves up to 46% of the execution time in some of the benchmarks.

2. Write-set-entry-caching: every transactional read or write of a value causes a lookup in the transactional log (implemented with a hashtable). We eliminate these lookups by caching the STM’s state at the value: ‘int x’ becomes ‘TxnValue<int> x’. As the log information is now stored directly at the value, no expensive lookups are needed. This makes some benchmarks run twice as fast.

3. Atomic tiling: if an atomic block is small and it is inside the body of a loop that is executed often, repeated transaction setup and commit add up to a large overhead. We increase the size of the atomic block by transforming the loop into a two-dimensional loop nest. If the original loop executes $N$ times, the newly created outer loop has $N/T$
iterations (for some default tile size $T$). Its body contains the atomic block holding an inner loop that iterates $T$ times what has been the original atomic block. Hence, this increases the amount of work performed by an atomic block. Atomic tiling saves up to 42% of the execution time on benchmarks that carry this coding pattern.

4. **Reducing the overhead of successive reads and writes**: operations on arrays inside a transaction are expensive because many elements may be accessed, possibly several times (repeated STM state lookups for the same array chunk). TransC can tile such loops into a two-dimensional loop nest with an inner loop of tile size $T$. In the inner loop, an aggregated transactional read copies all the array elements that the original loop reads into a local array. The new inner loop then only reads from and writes to local arrays. After the inner loop terminates another large combined transactional operation writes the local arrays back to the actual arrays. This transformation saves redundant transactional operations on the same array element and reduces transactional overhead by combining several reads/writes on sequential memory locations into one, which improves performance by a factor of 2 on some benchmarks.

5. **Independent transactions**: variables are by default mapped to locks in one large lock set of the STM. If two transactions access different variables that are accidentally mapped to the same lock (false sharing), one of the transaction cannot acquire all its locks and thus aborts. A programmer can mark atomic blocks as independent, i.e., that they only access disjunct data. We then allocate a different lock set for each block. For example, `atomic(x)` `{..}` causes the use of a lock set associated with object $x$. On benchmarks where it is applicable, this transformation saves up to 68% of the runtimes.

6. **Relationships between data structures**: STMs treat objects used in an atomic block as unrelated even if they are part of some bigger data structure such as a list. Consider two transactions that walk a list, where the first transaction changes field $f$ in the 1000th list element. If the second transaction reaches this element and reads $f$, it will be aborted. We use an annotation that puts the list elements and the list itself into a relationship. When a list element is changed, this is signaled to the list. When a transaction reads a list element, it first queries the list if an element was changed. In that case, the reading transaction is aborted. The signal operation corresponds to a transactional write to a dummy element $d$ the transformation adds to the list. The query is a transactional read that aborts the transaction if $d$ was written by another concurrently running transaction. With this transformation, the second transaction is aborted earlier in the above example, before it reaches the 1000th element. This wastes less time in transactions that are doomed to fail and saves up to 12% on benchmarks that use long lists.

**References**

Source Code Transformations to Increase the Performance of Software Transactional Memory

contribution
- six novel source code transformations that make better use of STM operations
- orthogonal to other STM optimizations
- portable to other STMs

1. Piggybacking

Goal: Protect contentious parts of a transaction against early aborts

Approach: Move statements at the end of an atomic block with compile-time fixed addresses to an extra function; execute that function within the commit phase while the necessary locks are held

void f() {
    cnt++;
}

void put(Item item) {
    atomic |
    int v = item.hashValue();
    list[v].add(item);
    cnt++;
}

2. Write-Set-Entry Caching

Goal: Reduce lookups in the log data structure for transactional operations

Approach: Store log information at the variables

\[
write \; x = 42 \Rightarrow \text{lookup/create/update the log entry in the transaction}
\]

3. Atomic Tiling

Goal: Increase the amount of work performed by atomic blocks in loops

Approach: Restructure the loop into a two-dimensional loop nest with the inner loop in the atomic block; loop to tile is annotated by the programmer

\[
\text{for} \{ \text{tile} \} \; i = 0 \; \text{to} \; N \{ \text{atomic} \{ \text{tile} \} \{ \}
\]

\[
\text{for} \; k = 0 \; \text{to} \; N \; \text{by} \; 4 \{ \text{atomic} \{ \}
\]

5. Independent Transactions

Goal: Prevent transactions that access disjunct data from mapping changed variables to the same locks at commit time

Approach: Let atomic blocks use different lock sets (annotated by programmer)

\[
\text{for} \; j = 0 \; \text{to} \; N \; \text{by} \; S \{ \text{atomic} \{ \}
\]

References