# Estimating Resource Bounds for Nested Transactions with Join Synchronization

Thi Mai Thuong Tran[1], Martin Steffen[1], and Hoang Truong[2]

[1] Department of Informatics, University of Oslo, Norway
[2] University of Engineering and Technology, VNU Hanoi

Software Transactional Memory (STM) has recently been introduced to concurrent programming languages as an alternative for lock-based synchronization, enabling an optimistic form of synchronization for shared memory. Supporting *nested* and *multi-threaded* transactions is an advanced feature of recent transactional models. Multi-threaded transactions mean that inside one transaction there can be more than one thread running in parallel. *Nesting* of transactions means that a parent transaction may contain one or more child transactions which must commit before their parent. Additionally, if a transaction commits, all threads spawned inside must join via a commit. To achieve isolation, each transaction operates via reads and writes on its own local copy of the memory, e.g., a local log is used to record these operations to allow validation or potentially rollbacks at commit time. The logs are a critical factor of memory resource consumption of STM. As each transaction operates on its own log of the variables it accesses, a crucial factor in the memory consumption is the number of thread-local transactional memories (i.e., logs) that may co-exist at the same time in parallel threads. Note that the number of logs neither corresponds to the number of transactions running in parallel (as transactions can contain more than one thread) nor to the number of parallel threads, because of the nesting of transactions. A main complication is that parallel threads do not run independently; instead, executing a commit in a transaction may lead to a form of implicit *join synchronization* with other threads inside the same transaction.

In this paper, we develop a type and effect system for statically approximating the resource consumption in terms of the maximum number of logs of a program. In the analysis, we use a variant of Featherweight Java extended with transactional constructs known as Transactional Featherweight Java (TFJ) The language features non-lexical starting and ending a transaction, concurrency, choice and sequencing. The analysis is *compositional,* i.e., syntax-directed. The analysis is *multi-threaded* in that, due to synchronization, it does not analyze each thread in isolation, but needs to take their interaction into account. This complicates the effect system considerably, as the synchronization is implicit in the use of commit-statements and connected to the nestedness of the transactions.