Estimating Resource Bounds for Software Transactions

Thi Mai Thuong Tran*, Martin Steffen, and Hoang Truong

Dept. of Computer Science, University of Oslo, Norway and University of Engineering and Technology, Vietnam National University of Hanoi

1 Motivation

Software Transactional Memory (STM) has recently been introduced to concurrent programming languages as an alternative for locked-based synchronization. STM enables an optimistic form of synchronization for shared memory. Each transaction is free to read and write to shared variables and a log is used to record these operations for validation or potentially rollbacks at commit time. Maintaining the logs is a critical factor of memory resource consumption of STM.

One of the advanced transactional calculi recently introduced is Transactional Featherweight Java (TFJ) [2], a transactional object calculus which supports *nested* and *multi-threaded* transactions. Multi-threaded transactions mean that inside one transaction there can be more than one thread running in parallel. *Nested* means that inside one transaction, there can be another transaction nested. Furthermore, nested transactions must commit before their parent transaction, and if a parent transaction commits, all threads spawned inside a transaction must join via a commit.

In this setting, a program execution may exceed the upper bound on the number of transactions the system can afford. Transactions contribute to the resource consumption which may lead to a memory overrun in the following way:

- duplicating parent transactions for the conflict checking. Each time a new thread is spawned, the log of its parent transaction is *copied* into the spawned thread's log. In other words, a spawned thread will "inherit" its parent transactions. So the resources for the new thread need to be calculated to store information in the parent transaction's log apart from its own log.
- a certain amount of transactions run in parallel at the same time which will increase the overall number of transactions in the system.

In this work, we will statically predict resource consumption in connection with transactions by identifying the maximum number of logs produced at any given point in time during the parallel execution of transactions. From that maximum, we can infer information about resource consumption such as memory usage.

* E-mail: tmtran@ifi.uio.no

$P ::=$	$\mathbf{0} \mid P \parallel P \mid p\langle e \rangle$	processes/threads
$L ::=$	$\mathsf{class}\ C\{\boldsymbol{f}{:}\boldsymbol{T}; K; \boldsymbol{M}\}$	class definitions
$K ::=$	$C(\boldsymbol{f} : \boldsymbol{T})\{\mathsf{this}.\boldsymbol{f} := \boldsymbol{f}\}$	contructors
$M ::=$	$m(\boldsymbol{x}{:}\boldsymbol{T})\{e\} : T$	methods
$e ::=$	$v \mid v.f \mid v.f := v \mid \texttt{if}\ v\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{let}\ \ x{:}T = e\ \texttt{in}\ \ e \mid v.m(\boldsymbol{v})$	expressions
	$\mid\ \texttt{new}\ \ C(\boldsymbol{v}) \mid \texttt{spawn}\ \ e \mid \texttt{onacid} \mid \texttt{commit}$	
$v ::=$	$r \mid x \mid \texttt{null}$	values

Table 1. Abstract syntax

2 A type and effect system for a transactional calculus

Syntax

The language used in this paper is, with some adaptations, taken from [2] and a variant of Featherweight Java (FJ) [1] extended with *transactions* and a construct for thread creation. The syntax of our calculus is given in Table 1. The main adaptations are: we added standard constructs such as sequential composition (in the form of the let-construct) and conditionals.

The language is multi-threaded: `spawn` e starts a new thread of activity which evaluates e in parallel with the spawning thread. Specific for TFJ are the two constructs `onacid` and `commit`, two dual operations dealing with transactions. The expression `onacid` starts a new transaction and executing `commit` successfully terminates a transaction.

Typing judgment

In order to estimate the maximal resource consumption used by an expression in the program, we introduce the judgments of the expressions as follows:

$$n_1 \vdash e :: n_2, h, l, t, S \tag{1}$$

The elements n_1, n_2, h, and l are natural numbers with the following interpretation. n_1 and n_2 are the pre- and post-condition for the expression e, capturing the nesting depth: starting at a nesting depth of n_1, the depths is n_2 after termination of e. We call the numbers n_1 resp. n_2 also the current balance of the thread. Starting from the pre-condition n_1, the numbers h and l represent the maximum resp., the minimum value of the balance during the execution of e (the "highest" and the "lowest" balance during execution). The numbers so far describe the balances of the thread executing e. During the execution of e, however, new child threads may be created via the spawn-expression and the remaining elements t and S take (also) their contribution into account. The number t represents the maximal, overall ("total") resource consumption during the execution of e, including the contribution of all spawned threads. The last component S is a multiset of pairs of natural numbers, i.e., it is of the form

$\{(p_1, c_1), (p_2, c_2), \ldots\}$. For all spawned threads, S keeps its maximal contribution to the resource consumption at the point after e, i.e., (p_i, c_i) represents that the thread i can have maximally a resource need of $p_i + c_i$, where p_i represents the contribution of the spawning thread ("parent"), i.e., the current nesting depth at the point when the thread is being spawned, and c_i the additional contribution of the child threads itself.

3 Main results

- We present a concurrent object-oriented calculus supporting nested and multi-threaded transactions. The language features non-lexical starting and ending of multi-threaded and nested transactions.
- We propose a type and effect system to guarantee safe commits and estimate the upper bound of resource consumption during its execution. This helps to predict the usage of resources in concurrent transaction systems.
- We show the soundness of the static analysis.

References

1. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '99*, pages 132–146. ACM, 1999. In *SIGPLAN Notices*.
2. S. Jagannathan, J. Vitek, A. Welc, and A. Hosking. A transactional object calculus. *Science of Computer Programming*, 57(2):164–186, Aug. 2005.