Estimating Resource Bounds for Software Transactions

Mai Thuong Tran, Martin Steffen, and Hoang Truong

University of Oslo, Norway
Vietnam National University, Việt Nam

Nordic Workshop on Programming Theory, NWPT2011,
Västerås, Sweden, October 26-28, 2011
Motivation

- (software) transaction as modern concurrency control mechanism
- proposed/being developed for a number of PLs
- a number of perceived advantages for user: enhanced performance + programmability
- price to pay: memory resource consumption
optimistic concurrency: not “prevent” potential future mutex violation at the entry of a CR, but check and potentially repair/compensate/undo (potential) conflicts at the end.

- conflict management (conflict detection + potential roll-back)
  ⇒ info to reconstruct the original state needs to be stored
Model: Transactional Featherweight Java

- TFJ: formal proposal for Java + transactions
- transactions model:
  - nested
  - multi-threaded
- “inheritance” of the resource consumption of parent thread
- child threads: joining commit ⇒ implicit synchronization ⇒ main complication
Nested and multi-threaded transactions
Nested and multi-threaded transactions
Nested and multi-threaded transactions

\[ I_1 : \log_1 \]
Nested and multi-threaded transactions

$l_1 \vdash \log_1$

$l_1 \vdash \log_1$

$l_2 \vdash \log_1$

$l_3$
Nested and multi-threaded transactions

\[ l_1 : \log_2 \]
Nested and multi-threaded transactions

$l_1: \log_2, l_2: \emptyset$
Nested and multi-threaded transactions

\[ l_1: \log_2, l_2: \log_3 \]
Nested and multi-threaded transactions

$l_1: \log_2'$
Nested and multi-threaded transactions

\[ l_3 \]

\[ l_2 \]

\[ l_1: \log_4' \]
TFJ syntax

\[ P ::= 0 | P \parallel P | p\langle e \rangle \quad \text{processes/threads} \]
\[ L ::= \text{class } C\{\vec{f}:\vec{T}; K; \vec{M}\} \quad \text{class definitions} \]
\[ K ::= C(\vec{f}:\vec{T})\{\text{this.}\vec{f} := \vec{f}\} \quad \text{constructors} \]
\[ M ::= m(\vec{x}:\vec{T})\{e\} : T \quad \text{methods} \]
\[ e ::= v | v.f | v.f := v | \text{if } v \text{ then } e \text{ else } e \quad \text{expressions} \]
\[ \quad | \text{let } x:T = e \text{ in } e | v.m(\vec{v}) \]
\[ \quad | \text{new } C(\vec{v}) | \text{spawn } e | \text{onacid} | \text{commit} \]
\[ v ::= r | x | \text{null} \quad \text{values} \]
Goal & complications

Goal

Static estimation on upper bound of resource consumption

- memory consumption = number of transactions potentially running at in parallel × local resource consumption

challenges

- “concurrent” analysis (≠ safe-commits ... iFM’10, FSEN’10 [2, 1])
- implicit join-synchronization via commits (≠ “Resource bounds for components” (ICTAC’05, FMOODS’05 [3, 4] ... )
- multithreading and nested transactions ⇒ parent-child relationship between threads relevant
Challenges

- compositional, syntax directed analysis

⇒: “interface information”

- e.g., “safe commit”:
  - “single threaded”: pre-and post are enough

\[
\begin{align*}
  n & \vdash \text{commit} :: n - 1 \\
  n_1 & \vdash e_1 :: n_2 \quad n_2 & \vdash e_2 :: n_3 \\
  \overline{} & \quad n_1 \vdash e_1; e_2 :: n_3
\end{align*}
\]

- counting components
Challenges

- **compositional**, syntax directed analysis
  ⇒: “interface information”
- e.g., “safe commit”:
  - “single threaded”: pre-and post are enough

\[ n \vdash \text{commit} :: n - 1 \]

\[ \vdash e_1 :: t_1 \quad \vdash e_2 :: t_2 \]

\[ \vdash e_1; e_2 :: t_1 \lor t_2 \]

- counting components
Challenges

- **compositional**, syntax directed analysis
  - $\Rightarrow$: “interface information”
- e.g., “safe commit”:
- counting components
  - $\parallel$ without synchronization
    \[
    \vdash P_1 :: t_1 \quad \vdash P_2 :: t_2 \\
    \frac{}{\vdash P_1 \parallel P_2 : t_1 + t_2}
    \]
  - $;$ *explicit* sequentialization/join
    \[
    \vdash P_1 :: t_1 \quad \vdash P_2 :: t_2 \\
    \frac{}{\vdash P_1;P_2 : t_1 \lor t_2}
    \]
Challenges

- compositional, syntax directed analysis
  ⇒: “interface information”
- e.g., “safe commit”:
- counting components
- here:
  - neither independent parallelism nor full sequentialization
  - implicit join synchronization via commits
    \[(\text{spawn } e_1); e_2\]
Joining commit

\begin{align*}
onacid; \\
onacid; \\
oacid; \\
\text{spawn } (e_1; \text{commit}^2) & \quad // \ 3 \\
oacid; \\
\text{spawn } (e_2; \text{commit}^3); & \quad // \ 4 \\
\text{commit; } \\
e_3 & \quad // \ 5 \\
\text{commit; } \\
e_4; & \quad // \ 6 \\
\end{align*}

in the following:
\begin{align*}
onacid \Rightarrow & \ [ \\
\text{commit } \Rightarrow & \ ] \\
e_1 & = \ [; [; [; \ldots ; ]; ]; ]; ] = [^3; \ldots ; ]^3 \\
e_2 & = \ [^4; \ldots ; ]^4 \\
e_3 & = \ [^5; \ldots ; ]^5 \\
e_4 & = \ [^6; \ldots ; ]^6 \\
\end{align*}
Joining commit


  e₁ — [ — ] —

  e₂ — [ — ]
Joining commit
Joining commit
Judgment & interface information

Judgment

\[ n_1 \triangleright e :: n_2, h, l, \vec{t}, S \]

- current thread
  - \( n_1 \) and \( n_2 \): balance, pre- and post-condition
  - \( h, l \): high- and low-point during execution
- not (only) current thread
  - \( \vec{t} \): sequence of total weights of current + spawned threads, separated by joining commits
  - \( S \): contribution of spawned threads after execution of \( e \)
Sample derivation: pre- and post

\[
0 \vdash [ [ ; \text{spawn} \left( e_1 \right) ] ] :: 2
\]

\[
2 \vdash [ ; (\text{spawn} \left( e_2 \right) ) ] ; ] ; e_3 \} ; e_4 :: 1
\]

\[
0 \vdash [ [ ; \text{spawn} \left( e_1 ; \right) ] ] ; [ ; (\text{spawn} \left( e_2 ; \right) ) ] ; ] ; e_3 \} ; e_4 :: 1
\]

\[
\begin{array}{c}
\text{e}_1 \\
\text{e}_2 \\
\text{e}_3 \\
\text{e}_4
\end{array}
\]

\[
\begin{array}{c}
n = 0 \\
n = 2 \\
n = 2 \\
n = 1
\end{array}
\]
Sample derivation (high and low)

\[
0 \vdash [ [ ; \text{spawn} (e_1) ] ] :: 2, 0
\]
\[
2 \vdash [ ; (\text{spawn} (e_2) ) ] ; ] e_3 ; e_4 :: 7, 1
\]
\[
0 \vdash [ [ ; \text{spawn} (e_1) ] ] ; [ ; (\text{spawn} (e_2) ) ] ] ; ] e_3 ; e_4 :: 7, 0
\]
Sample derivation (par. contribution and synchronization)

\[ 0 \vdash \left[ \left[ ; \text{spawn} \left( e_1 \right) \right] \right] :: [7], \{(2, 3)\} \]

\[ 2 \vdash \left[ \left[ ; \left( \text{spawn} \left( e_2 \right) \right) \right] \right]; \left[ ; \left( \text{spawn} \left( e_3 \right) \right) \right] ; e_4 :: [10, 8], \{(1, 0)\} \]

\[ 0 \vdash \left[ \left[ ; \text{spawn} \left( e_1 \right) \right] \right]; \left[ \left[ ; \left( \text{spawn} \left( e_2 \right) \right) \right] \right]; \left[ ; \left( \text{spawn} \left( e_3 \right) \right) \right] ; e_4 :: t, \{(1, 0), (1, 0)\} \]

\[ t = 7 \lor (10 + |\{(2, 3)\}|) \lor (8 + |\{(1, 0)\}|) \]

\( n = 0 \quad n = 2 \quad n = 2 \quad n = 1 \)
Sequential composition

\[ n_1 \vdash e_1 :: n_2, h_1, l_1, \vec{s}, S_1 \quad n_2 \vdash e_2 :: n_3, h_2, l_2, \vec{t}, S_1 \]

\[ h = h_1 \lor h_1 \quad l = l_1 \land l_2 \]

\[ \vec{s} = s_1, \ldots, s_k \quad \vec{t} = t_1, \ldots, t_m \quad k, m \geq 1 \quad p = n_2 - l_1 \]

\[ t'_1 = t_1 + |S_1| \quad t'_2 = t_2 + |S_1 \downarrow n_2 - 1| \quad t'_3 = t_3 + |S_1 \downarrow n_2 - 2| \quad \ldots \]

\[ S_1^{\text{rest}} = S_1 \parallel \downarrow l_2 \quad S = S_1^{\text{rest}} \cup S_2 \]

\[ \vec{u} = s_1, \ldots, s_{k-1}, s_k \lor t'_1 \lor \ldots \lor t'_p, t'_{p+1}, \ldots, t'_m \]

\[ n_1 \vdash e_1; e_2 :: n_3, h, l, \vec{u}, S \]
Future work

- more fine-grained model
- towards a hybrid model
- higher-order functions
Safe locking for multi-threaded Java with exceptions.  

Safe commits for Transactional Featherweight Java.  
In D. Méry and S. Merz, editors, Proceedings of the 8th International Conference on Integrated Formal 
Methods (iFM 2010), volume 6396 of Lecture Notes in Computer Science, pages 290–304 (15 pages).  
and appeared as extended abstract in the Proceedings of NWPT’09.

Guaranteeing resource bounds for component software.  
In M. Steffen and G. Zavattaro, editors, FM OODS ’05, volume 3535 of Lecture Notes in Computer Science, 

Finding resource bounds in the presence of explicit deallocation.  