Statically Computing Upper Bounds on Object Calls for Pessimistic Concurrency Control

Konrad Siek
Poznań University of Technology
Konrad.Siek@cs.put.poznan.pl

Paweł T. Wojciechowski
Poznań University of Technology
Pawel.T.Wojciechowski@cs.put.poznan.pl

Abstract
Atomic RMI is a library for pessimistic concurrency control in distributed systems that uses versioning algorithms. These algorithms require a priori knowledge about execution of transactions to achieve maximum efficiency, i.e. the maximum number of times specific objects will be accessed. This paper presents an algorithm and a tool that establishes these upper bounds on objects accessed through static program analysis.

1. Introduction
The unpredictable manner in which concurrent data access is performed in shared memory systems causes concurrency control mechanisms continuously to gain importance, especially with the mounting popularity of multi-core processors. Therefore it is usual for programming languages to supply a set of constructs that facilitate the synchronization of concurrent processes, among them critical sections, monitors, locks, conditional variables, and semaphores. Nonetheless, these mechanisms are considered to be difficult to use and the developer must manage the blocking mechanisms unaided and on a low-level, as well as identify the sections of code that require synchronization—both being error-prone tasks. In addition, the low-level constructs are neither directly re-usable nor composable [10] (if every single component is safe by itself, it does not necessarily follow that their combination is safe).

Many authors have recently proposed to solve the problems of concurrent programming by turning to the idea of Software Transactional Memory (STM) [27]: STM allows the declaration of atomic transactions in concurrent programming (similar to transactions known from Database Management Systems) whose correct processing is enforced by the system, thus ensuring synchronization. Several such systems have been designed and implemented, though mostly non-distributed (see e.g. [10–13, 24, 26] Among others). Another line of research is on type systems for specifying and verifying the atomicity of methods in multithreaded programs (see [6] and follow-up papers). Roughly, it is guaranteed that the effect of an execution (i.e. the state of the system) of two or more concurrent atomic blocks of code will be equivalent to the effect of a sequential execution of those blocks.

In our work, we are interested in extending the idea of STM to distributed systems. In distributed systems, when two or more clients want to access shared remote resources consistently, it is necessary to control concurrent access to these resources. Since the distributed system can be loosely coupled and shared resources can be whatsoever, we exclude heavy-weight solutions based on distributed database systems. Several authors proposed Distributed STM, meaning a system with locally-scoped transactions executing on different network nodes, with shared data replicated on these nodes and kept consistent (see e.g. [3, 34] among others). In general, Distributed STMs can be seen as distributed shared memory systems (implementing distributed global address space), extended with the notion of atomic transactions. Alternative approaches include distributed transactions that can span many nodes; they are aimed at loosely-coupled distributed systems, such as the Internet. We consider STMs and distributed transactions that are object-based, in the sense that transactional synchronization works by intercepting and synchronizing calls to object methods, leveraging the infrastructure provided by object-oriented languages. An example of an object-based STM is DSTM2 [13], while JSTM [34] can be seen as an object-based Distributed STM.

We develop Atomic RMI [19, 32]—an object-based concurrency control library built on top of Java RMI which ensures linearizable distributed objects in Java by allowing the developer to define specific sequences of method calls on remote objects as distributed transactions. The library uses pessimistic concurrency control algorithms that suspend (block) methods calls when necessary to satisfy linearizability. We are currently extending our algorithms and implementation to support a rollback mechanism. Thus, the programmer will be offered a mechanism equivalent to distributed atomic transactions, in which shared data can be any objects defined as ‘remote’ (in the sense of Java RMI) and accessed by any of their methods. We have already designed and implemented a non-distributed variant of our library that supports rollback [17].

The result of the use of Atomic RMI, is lifting the chore of manual management of blockades from the developer, therefore reducing his efforts to declaring distributed transactions. However, the benefits in ease of use are offset by the requirements of the algorithms used by Atomic RMI. In order to schedule the execution of method calls, the algorithms require some input data about a transaction to be known a priori—i.e. before the transaction is spawned. More precisely, the developer needs to find the number of times a particular object will be accessed during run-time, or at least an upper bound on that number.

This paper proposes a method to gather the upper bounds on the number of calls to specific objects by static analysis in order to aid the programmer in using Atomic RMI transactions. This method was implemented in the form of a tool that is meant to be an integral part of Atomic RMI. However, our algorithm could also be useful in Distributed STMs and non-distributed STMs that use pessimistic concurrency control.
version number. The key idea is that an object method called
With respect to concurrency control, Atomic RMI uses versioning
conclusions are presented. Next, similar
simistic concurrency control in Atomic RMI are described. Then,
execution of distributed transactions nor STM (or Distributed
of programs that aims at deriving information about execution pat-
Figure 1. Transaction scheduling: (a) BVA, (b) SVA, and (c) RVA
There is a large body of research related to worst case analysis
programming asks of some these in Section 5). However,
we do not know examples of using this information for optimizing
the execution of distributed transactions nor STM (or Distributed
STM). Our paper shows how the information gathered from trans-
sectional code at compile time can make the execution of distributed
transactions in Atomic RMI more efficient. However, we think that
our approach is general enough to be applicable in the implemen-
tations of STM too.

The paper has the following structure. First, the concepts of pes-
simistic concurrency control in Atomic RMI are described. Then,
the algorithm and tool for static analysis are shown. Next, similar
work on worst case analysis of programs is described. Finally, our
conclusions are presented.

2. Concurrency Control in Atomic RMI
With respect to concurrency control, Atomic RMI uses versioning
algorithms [30, 31, 33]. The key idea is that an object method called
by a transaction is executed if that transaction holds a matching version number of the object. Otherwise, the call is blocked (put
to sleep), waiting passively for the version upgrade. Thus, version
numbers determine the order of executing object methods by trans-
actions; this order agrees with the isolation property. This simple
mechanism protects objects from being accessed by transactions
that—in order to satisfy the isolation property—should wait till
other transactions access these objects.

Figure 1(a) shows transaction scheduling using the BVA version-
ing algorithm. Note that a call to critical method a.m of trans-
action T3 is postponed until transaction T1 has completed. This
gives an almost serial execution of transactions. However, some
other critical method p of another (non-shared) object c can be ex-
cuted by transaction T2 in parallel with transaction T1; note that
c.p is executed by a separate thread that belongs to transaction T2
(e.g. it can be another RPC call to the remote object). The isolation
property is still satisfied since the effects of executing c.p cannot
be observed by transaction T1.

Figure 1(b) shows transaction execution using the SVA algo-

3. Finding Upper Bounds on Method Calls
We designed an algorithm, dubbed the Object Call Count Analysis
(OCCA), to count upper bounds on the number of calls issued
to each of a predefined set of objects. The algorithm statically
analyzes source code in the Simple Language (SL), a small subset
of the Java language. SL consists of only the most basic constructs,
like variable declaration, assignment, if statements, while loops,
wait; blocking can be implemented by exploring the partial infor-
mation about transaction execution. Figure 1(c) shows an example
execution of two transactions, T1 and T2, under control of the RVA
algorithm. For each transaction, the algorithm must know a priori,
a causal relation between object calls made by the transaction.

As the example indicates, in relation to BVA, the RVA and
SVA can be expected to shorten the system’s response time (the
time before a transaction terminates) and increase its capacity for
simultaneously running transactions. We are mostly concerned with
SVA at this time, although extending our efforts to encompass RVA
may prove to be an interesting topic for future work.

PROPOSITION 1 (Supremum). The OCCA algorithm computes
the upper bounds (supremum) on objects’ method calls for any
correct program in the SL language. All method invocations in
code (that are not dead code) are considered, and the estimated
upper bound is never lower than the actual number of method inv-
ocations.

PROPOSITION 2 (Safety). The OCCA algorithm is safe: it cannot
produce any upper bound on the number of object accesses that is
lower than the actual number of this objects’ method calls during
any execution of the analyzed program.

PROPOSITION 3 (Termination). The OCCA finishes in a finite
number of steps, even on occurrence of recursion or infinite loops.

The algorithm’s input takes the form of source code in SL
expressed as an expression tree. The algorithm’s output is a map
where keys are unique object identifiers pointing upper bounds on
the number of times the objects were used to make a method call.

1 Our implementation supports a subset of Java that is a superset of SL, and
therefore is more fit for use.
This number can be zero, any finite number, or \( \omega \)—the symbol representing an infinite or unpredictable result.

The algorithm is started by the application of function \texttt{analyze_expr}, which takes the expression tree \( E \) and the number of executions of this fragment of code, as arguments. The function then decides what particular type of expression it was provided with, and then delegates the analysis to one of the specialized functions. The function is given in Figure 2. The type of the main function and the specialized functions is defined as:

\[
\text{analyze_expr} : \text{expression} \rightarrow (\mathbb{N}_0 \cup \omega) \rightarrow \{\text{id} \rightarrow \mathbb{N}_0 \cup \omega\}
\]

where \( \text{id} \) belongs to the set of all possible identifiers.

The most interesting aspects of the analysis are handling of loops, conditional expressions, and method calls, and these are explained below. Detailed information can be found in TR [28].

### 3.1 Method call analysis

When a method call has been found in the code, the function indicates that the object implementing the analyzed method will be called at most a certain number of times. The function returns a single-element map: if the analyzed expression is evaluated only once, then the object identifier will be mapped to the value of one; if the expression may be evaluated more times (e.g. in a loop) then the number of evaluations is included in the map. Later, the number from all such maps will be totaled to produce a global estimate.

If the body of the analyzed method is known, then the method’s arguments and its body are further analyzed. Additionally, every time a method’s body is analyzed, the global visit counter for that method is incremented, and if the counter exceeds a certain threshold, it must be assumed that infinite recursion occurred, and no more analysis of that method’s body should be allowed.

In cases where the body method is not analyzed, the states of values that may be potentially modified within the body should be set to unknown. If the body is of the method known, the variables that need to be subjected to this may be derived by analysis.

The function can be expressed formally as,

\[
\text{analyze_call}(E, n) = \begin{cases} \{\text{object}(\text{variable}(E)) \mapsto n\}, & \text{if } \text{has_body}(E) \\ \text{add}(\text{object}(v) \mapsto n), & \text{if } V < M \\ \text{analyze_block}(P \cup B, n), & \text{if } \text{has_body}(E) \\ \text{object}(\text{variable}(E)) \mapsto \omega, & \text{if } V \geq M \end{cases}
\]

where

\[
v = \text{variable}(E), B = \text{method_body}(E), \\
P = \text{method_argument_assignments}(E), \\
V = \text{visited}(E), M = \text{MAX_ITERATIONS}
\]

### 3.2 Conditional block analysis

The function attempts to evaluate the guard condition whose outcome may be either \textit{true}, \textit{false}, or \textit{uncertain}. In the foremost case the positive block is then further analyzed, while in the case of evaluation to \textit{false} the positive block is ignored, and if the alternative block is analyzed if it is defined. If the guard condition is not a literal, but another expression, it must also be analyzed.

In the case of \textit{unpredictable}, both blocks must be analyzed further and the maximum of both analyses must be derived using \texttt{join}. In addition, if there are assigned values that differ between blocks, the variable must be noted to have an unknown value.

The \texttt{join} operation \( \vee \) takes two maps and returns a new map, which contains all key-value pairs from both maps. If a key is present in both input maps, the referenced values are compared, and the higher of the two is used in the output map. The \texttt{join} operator ensures that the maximum number of method calls will be selected as a result of the analysis.

The conditional analysis function is defined as follows,

\[
\begin{align*}
\text{analyze_indicator_block}(E, n) = & \begin{cases} \\
\text{analyze_proper}(E, C, n) & \text{if } \text{is_literal}(C) \\
\text{add}(&\text{analyze_expr}(C, n), \\
\text{analyze_proper}(E, C, n)) & \text{if } \neg \text{is_literal}(C) \\
\end{cases} \\
\text{analyze_proper}(E, C, n) = & \begin{cases} \\
\text{analyze_expr}(E_{\text{positive}}, n) & \text{if } \text{is_true}(C) \\
\text{analyze_expr}(E_{\text{alternative}}, n) & \text{if } \text{is_false}(C) \\
\text{analyze_expr}(E_{\text{positive}}, n) & \text{if } \neg \text{is_true}(C) \\
\text{analyze_expr}(E_{\text{alternative}}, n) & \text{if } \neg \text{is_false}(C) \\
\text{add}(&\text{analyze_expr}(E_{\text{positive}}, n), \\
\text{analyze_expr}(E_{\text{alternative}}, n)) & \text{if } \neg \text{is_true}(C) \\
\text{analyze_expr}(E_{\text{alternative}}, n) & \text{if } \neg \text{is_false}(C) \\
\end{cases} \\
\text{where} & \\
C = \text{condition}(E), \\
E_{\text{positive}} = \text{positive_block}(E), \\
E_{\text{alternative}} = \text{alternative_block}(E)
\end{align*}
\]

### 3.3 Loop block analysis

The function attempts to analyze an expression \( E \) representing a loop. The loop’s condition is evaluated: if the result is \textit{true} both the condition \( C \) and the body \( B \) of the loop are further analyzed before the next round of loop analysis is conducted; whereas if the result is \textit{false} only the condition is analyzed and the analysis of the loop ends. If the condition evaluates to neither \textit{true} nor \textit{false} but is indefinite, the condition and the body are both analyzed using unknown values.

Since it is possible for the analysis never to terminate if the condition always evaluates to \textit{true}, the parameter \( i \) tracks the number of iterations, and when it exceeds a threshold the analysis assumes the loop will run indefinitely and treats the loop as if the condition evaluated to an unknown value. If the condition of the loop is unknown, an analysis is performed, which marks all of the values that may be written to as unknown.

\[
\begin{align*}
\text{analyze_loop_block}(E, n) = & \begin{cases} \\
\text{analyze_loop}(E, n, \text{MAX_ITERATIONS}) & \text{if } C = \text{condition}(E), \\
\text{analyze_loop}(E, n, i) = & \begin{cases} \\
\text{add}(&\text{analyze_expr}(C, n), \\
\text{analyze_expr}(B, n)) & \text{if } \text{is_true}(C) \land i > 0 \\
\text{analyze_expr}(E, n, i - 1) & \text{if } \text{is_false}(C) \land i > 0 \\
\text{add}(&\text{analyze_expr}(C, n), \\
\text{analyze_expr}(E, n, i - 1)) & \text{if } \neg \text{is_true}(C) \vee i \leq 0 \\
\text{analyze_expr}(B, n) & \text{if } \neg \text{is_false}(C) \vee i \leq 0 \\
\end{cases} \\
\text{where} & \\
C = \text{condition}(E), B = \text{body}(E)
\end{cases}
\end{align*}
\]

This approach is resource-intensive, so a more light-weight alternative is considered: an auxiliary function \texttt{iteration_count} predicts the maximum number of loop iterations beforehand, then the condition and the body of the loop are analyzed, with the iteration count as an argument.

\[
\begin{align*}
\text{analyze_loop_block}(E, n) = & \begin{cases} \\
\text{add}(&\text{analyze_expr}(C, n + m), \\
\text{analyze_expr}(B, n + m)) & \text{if } C = \text{condition}(E), B = \text{body}(E), \\
\end{cases} \\
\text{where} & \\
m = \text{iteration_count}(E)
\end{align*}
\]

A simple and imprecise function establishing the maximum number of loop iterations can determine if the loop will be evaluated at all, in which case it returns 0, and otherwise returns the unknown value \( \omega \). The conception of better functions is widely covered in [2, 4, 7, 8].
analyze_expr(E, n) = 
\begin{align*}
\text{analyze_declaration(child(E), n)} \\
\text{analyze_initialization(child(E), n)} \\
\text{analyze_assignment(child(E), n)} \\
\text{analyze_call(child(E), n)} \\
\text{analyze_block(child(E), n)} \\
\text{analyze_conditional_block(child(E), n)} \\
\text{analyze_loop_block(child(E), n)}
\end{align*}
\text{if type}(E) = \text{variable_declaration} \\
\text{if type}(E) = \text{object_initialization} \\
\text{if type}(E) = \text{variable_assignment} \\
\text{if type}(E) = \text{method_call} \\
\text{if type}(E) = \text{block} \\
\text{if type}(E) = \text{conditional_block} \\
\text{if type}(E) = \text{loop_block}
\) (2)

4. Code Generation
A prototype static analyzer was built based on the algorithm described above. Apart from implementing the various analyses, the tool generates preambles to transactions that it finds in the code, with the upper bounds defined using the Atomic RMI API.

The code generation itself is done by injecting tokens into the processed code on the lexical level. Before it is done, the appropriate places in the token stream must be found, and this is done during the analysis of the expression tree.

5. Related work
A large body of work is available on the use of static analysis in the Worst-Case Execution Time problem [16, 29], where methods like path analysis [9], symbolic analysis [18], and partial evaluation [14] are employed to establish upper bounds on the time code will take to run. Emphasis is placed on finding in advance the maximum number of loop iterations [2, 4, 7, 8]. Frameworks for WCET are available, like aiT [5] or Bound-T [15].

This work also bears resemblance to finding user-definable usage bounds (for Java [20, 21, 23] in particular, but also other languages [22]) where prediction of the maximum number of uses of particular resources is attempted by code analysis.

In the context of code generation, a number of tools employ static analysis to deduce properties of the code that the following generation requires, e.g. AutoBinder [25] a Python binding generator for the C language, or Jass [1] (Java with Assertions), a tool for injecting assertions based on defined invariants.

6. Conclusions
The investigation of the subject has led to the conception of the algorithm, as well as the implementation of a prototype tool for static analysis that is able to process a subset of the Java language (extended in relation to SL) and generate safe, execution-ready code. The tool, therefore, relieves the developer of having to create these preambles manually.

Both the tool and the algorithm itself are designed with extension in mind, to allow future expansion beyond the limited capabilities of the SL, as well as to refine the analyses to produce more precise bounds. Specifically, future work could involve improving the method for handling loops.

In addition, support for RVA in Atomic RMI may become a topic for future work, resulting in still improved response times.

References


