Analysis and Transformations for Efficient Query-Based Debugging

Michael Gorbovitski, Tuncay Tekle

Tom Rothamel, Scott Stoller

Y. Annie Liu

Computer Science Department
State University of New York at Stony Brook

Query-Based Debugging

allows powerful queries to be used in debugging
to monitor conditions and trigger actions as program executes

- may use all values in state, not only values in a single scope, and use even history of states

- makes debugging much easier

- is much more expensive to compute
values depended on change continuously as program executes

Overview of This Work

framework:

powerful queries, over all objects at any point and in history.
may contain joins, over sets and extents, and may be nested.

transformations:

incrementally maintain results of expensive queries at updates.
collect history information.

analysis:

alias analysis and type analysis are used for detecting updates.
must handle complex objects and data types.

experiments:

confirmed ease of use, and efficiency and effectiveness.
found all injected bugs and an actual bug in an FTP client.

Related work

QBD, impl: dynamic qbd [Lencevicius Hölzle Singh ECOOP99 ASE03], snapshot qbd [Potanin Noble Biddle AusSE04], PTQL/PARTIQLE [Goldsmith O'Callahan Aiken OOPSLA05], PQL [Martin Livshits Lam OOPSLA05], InvTS [OOPSLA05], JQL [Willis Pearce Noble ECOOP06], caching & inc [OOPSLA08], generating inc impl [GPCE08], runtime inv check [WODA08].
less powerful queries or non-incremental impl, except for last

AOP: [Kiczales et al ECOOP01 overview, Avgustinov et al AOSD06, ...].
manually maintain query results, lack powerful analyses

alias analysis: we extend best flow-sensitive analysis [Goyal HOSC05] to interprocedural, OO, for Python, and with a form of context sensitivity.

type analysis: we use iterative analysis [Eifrig Smith Trifonov OOPSLA95] and support multiple possible types of an expression and constant propagation in data structures.

Example: Violation of Invariant

valid-parent in trees: a child's parent field must point to its parent

```
foreach (n in extent(Node), m in extent(Node):  -- query
        m in n.children, n != m.parent):

   report(m, " is a child of ", n, ", but ",    -- action
          n, " is not the parent of ", m, "!")
   stop()
```

meaning: at any program point,
for each element in query result, do action

hard to write code to compute query result efficiently:
query has a join and is over two changing sets
result may be affected by any `x.add(y)`, since maybe `x=z.children`

Example: Violation of Temporal Property

nftp synchronization debugging (uncopied dirs): no new `ls` commands are sent when there are outstanding `cwd` commands

```
foreach (c1 in $exec_cmds, c2 in $exec_cmds:
         c1.cmd == 'ls', c2.cmd == 'cwd', c1.host == c2.host):
   report('ls and cwd are executed at the same time!')
   stop()

de in global: $exec_cmds=set()  -- declaration, for recording history

at $x.cwd($dir):                -- recording history
if type($x) == ftplib.FTP:
do before:
   $c=Cmd($x,'cwd')
   $exec_cmds.add($c)
do after:
   $exec_cmds.remove($c)
                                -- similar at $x.list() for cmd 'ls'
```

important to query easily using values from history of execution

Example: Cause of Exception

index out of bound: line 123, file t.py. find inevitable cause location

```
foreach (c in $C, i in $I: c.val != None, i.val != None):
   if outOfRange(c.val, i.val):
      if (c.val,i.locId) not in $bad: $bad[c.val,i.locId]=$LOCATION
   else:
      if (c.val,i.locId) in $bad: del $bad[c.val,i.locId]
de in global:
   $C,$I,$bad = set(),set(),dict()      -- sets of last update places
   def wrapper(L,R,locIdR):
      try: return L[R]
      except IndexError:
        report("Became innevitable at: ", bad[L,locIdR])
        stop()
var $L, $R
at $L[$R]:                              -- at exception, report cause
if line(123) and file('t.py')
do instead:
   wrapper($L,$R,locId('$R'))
at $e:                                  -- at update to R, maintain I
if part($e,'$x','alias($x,$R) and updates($x)')
do before:
   $obj=Update(locId=locId('$x'),val=$x)
   $I.discard($obj)
   $I.add($obj)                         -- similar at update to L
```

Debugging Rules

general form:

$$
\begin{array}{l}
\textbf{foreach } (\textit{query}) : \\
\quad \textit{action} \\
(\textbf{at } \textit{update} \\
\quad (\textbf{if } \textit{condition})? \\
\quad (\textbf{de } (\textbf{in } \textit{scope } (\textit{field}|\textit{method})^{+})^{+})? \\
\quad (\textbf{do } (\textbf{before}|\textbf{after}|\textbf{instead } \textit{maint})^{+})? \\
)*
\end{array}
$$

$query$ has the form $(v_1$ `in` $S_1, ..., v_k$ `in` $S_k : cond_1, ..., cond_j)$

$cond_i$ is e_1 op e_2, op is `==`, `!=`, `in`, `not in`, e_i is v or $v.f$,

or bool exp on objs in S_i, their fields, & immutable objs

Alias Analysis

take best intraprocedural flow-sensitive analysis for C, $O(NV^2)$
by Goyal [HOSC05]. N, V: num of nodes, vars in program.

extend to interprocedural for OO languages, $O(NV^2)$
rename all vars and fields to be distinct;
add and replace appropriate nodes and edges for call, return,
object creation, member function call, and field access

add a form of context-sensitivity, $O(N^2V^2)$
copy function for each call, but use same names for local vars

time complexity: $O(N^2V^2)$
observed quadratic in experiments

Type Analysis

use iterative analysis by Eifrig Smith Trifonov [OOPSLA95]

support multiple possible types of an expression
e.g., union(int(1),int(2))

do constant propagation for complex data types
e.g., if y=range(x), int(p) in type(x),
then list(int(i): i=0..p-1) in type(y)

time complexity: $O(NS)$. S: max num of vars in scope at a node.
observed linear in experiments

Experiments

qdbPy: prototype implementation of QBD for Python
compile debugging rules to InvTS rules, to instrument progs

find violation of invs, of temp properties, causes of exceptions
XML DOM transformations(10m elms) and an FTP client(bug)

slowdown in running times of applications: timeout without inc,
overhead 109-805% without analyses, 67-85% with all analyses

running times of instrumentation: for 493–15955 CFG nodes,
under 25 sec without analyses, under 1 min with all analyses

Running Times of Applications

1: lxml–valid parent, 2: lxml–no shared child and no own child

3: nftp–wait till commands complete, 4: lxml–exception cause

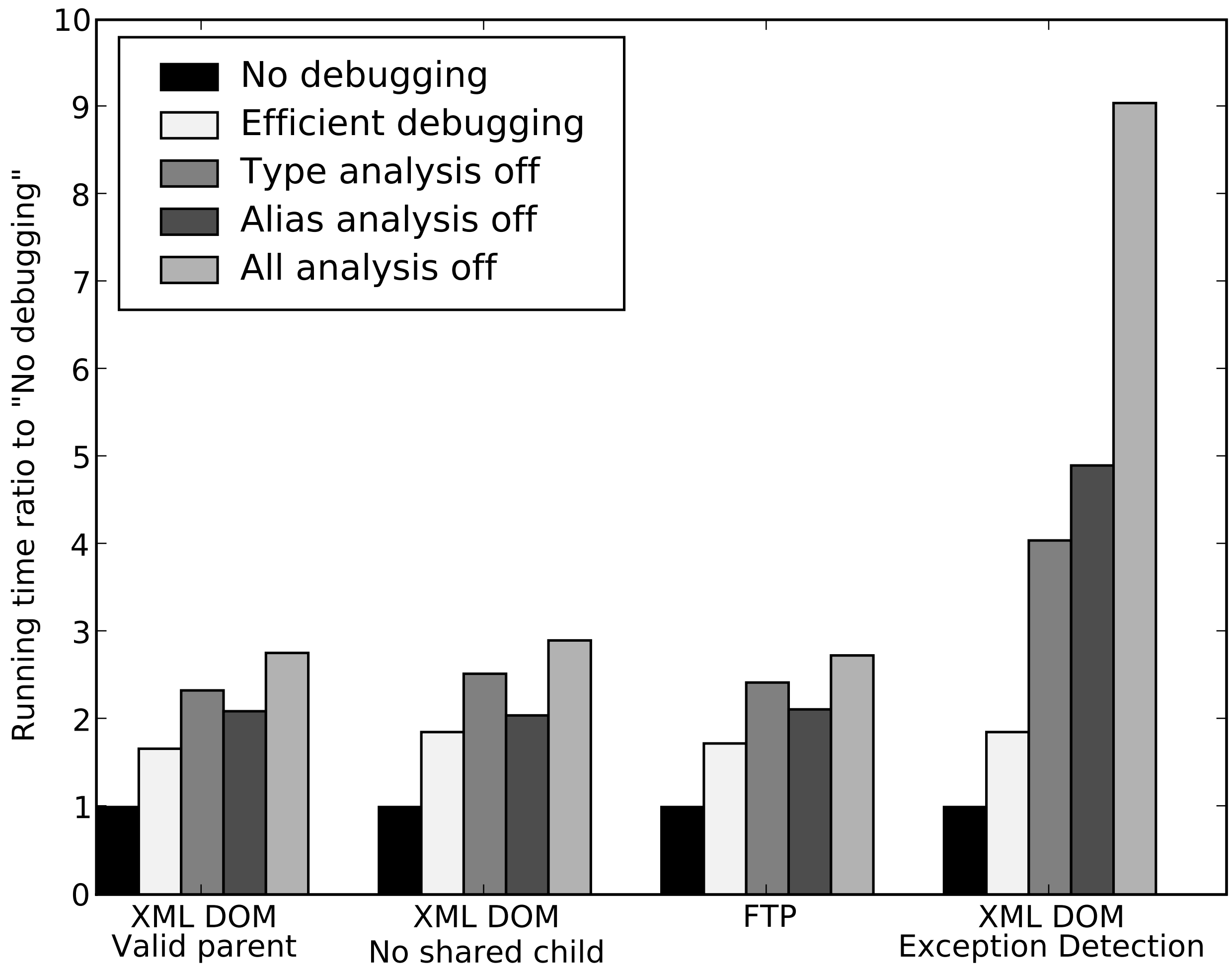
running times in seconds:

	no debugging	all analyses	no type analysis	no alias analysis	no analyses
1	21	35	49	44	58
2	21	39	53	43	61
3	326	563	790	690	891
4	21	39	85	103	190

timeout after 20 minutes without incrementalization—quadratic

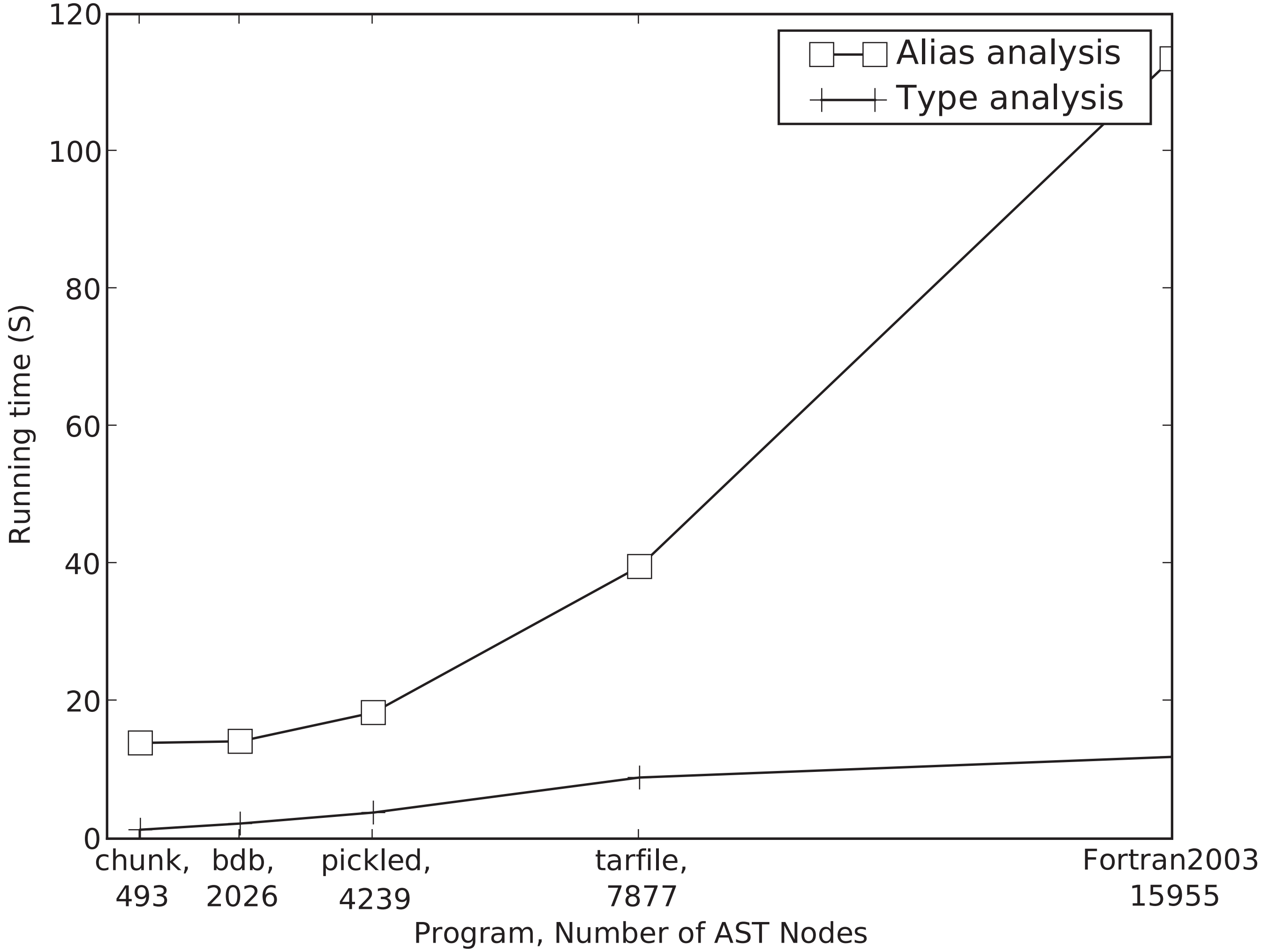
e.g., benchmark programs for lxml has 10 million elements

Slowdown in Running Times of Applications

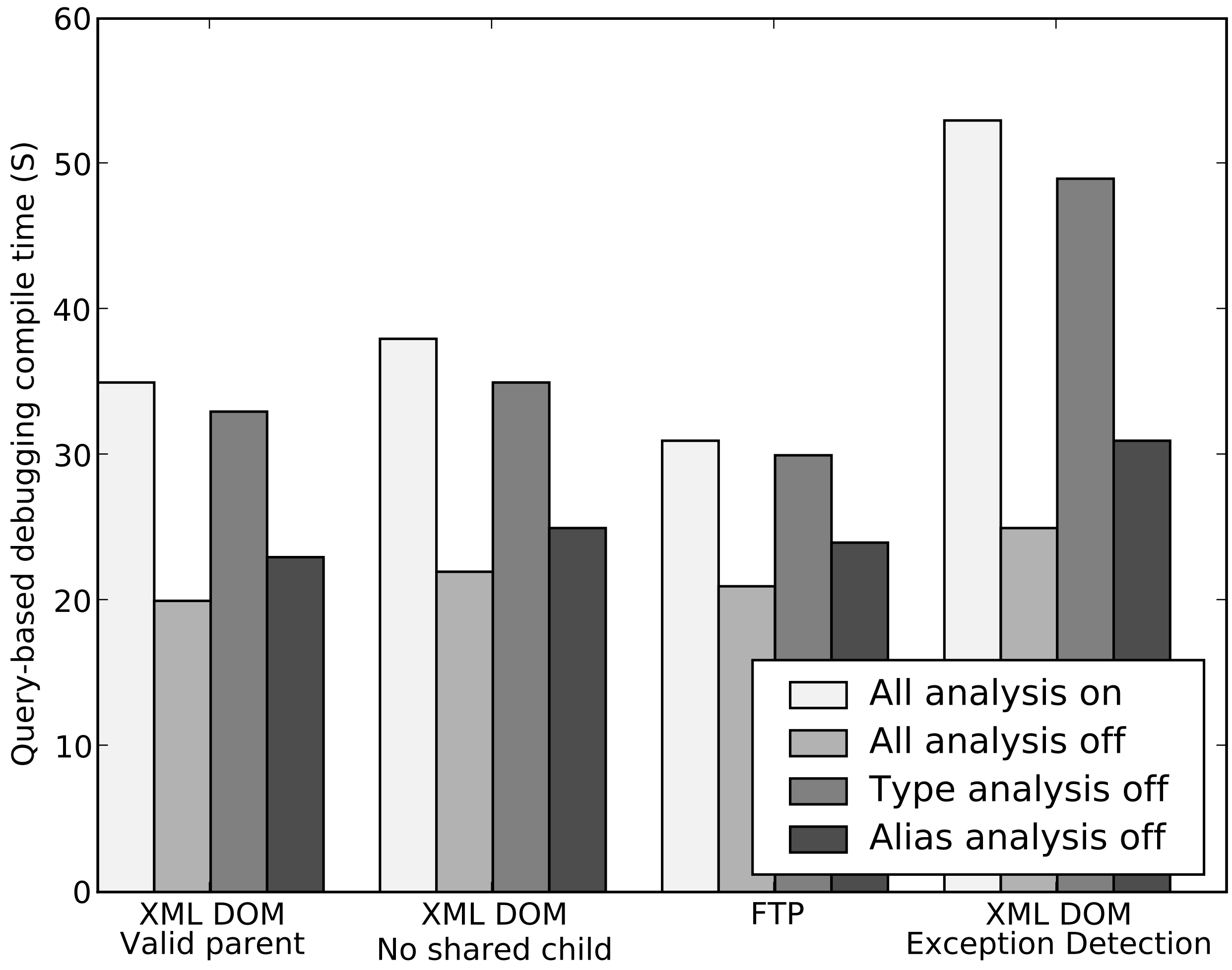


running times normalized to those with no debugging

Running Times of Alias and Type Analyses



Running Times of Instrumentation



Conclusion

- a framework that allows powerful queries in debugging
  may contain joins, over sets and extents, in state and history

- analyses and transformations for efficient implementations
  incrementalization, alias analysis, type analysis

- prototype and experiments
  confirm ease of use, efficiency, and effectiveness

- current and future work: use InvTS invariant rules for optimization (invariant maintenance), runtime verification (invariant checking), reverse engineering (invariant detection)