Incrementalization Across Object Abstraction

Y. Annie Liu

Computer Science Department
State University of New York at Stony Brook

joint work with
Scott Stoller, Michael Gorbovitski,
Tom Rothamel, and Ellen Liu

Object abstraction

encapsulation of data and operations: separate what from how.

incarnations:

abstract data types, objects and classes, components.

advantages:

enable construction of complex software systems by assembling software components.

facilitate program understanding, reuse, enhancement, etc.

raising the level of abstraction:

operations on bits, bytes, numbers, structured data, sets.

What

what users do in information processing/knowledge engineering:

queries: compute information using data w/o changing data.

updates: change data.

example:

class `LinkedList` in Java has many methods:

`size()`, 11 add or remove, several other queries.

How

how to implement the queries and updates: varies significantly

straightforward:

queries compute requested information.

updates change base data.

example: size() contains a loop that computes the size.

observe:

queries are often repeated, many are easily expensive;

updates can be frequent, they are usually small.

sophisticated:

store derived information; queries return stored information.

updates also update stored information.

example: maintain size in a field, and update it in 11 places.

Conflict between clarity and efficiency

straightforward: clear and modular, but poor performance.

sophisticated: good performance, but not clear or modular.

clarity and modularity
software productivity and cost $\longleftrightarrow$ system performance

much worse for complex systems: many queries and updates; queries may cross components; updates may be spread in many components.

Conflict — some more examples

role-based access control: secure access of resources

queries: check access, various review functions, ...

updates: add/delete user/role/session, grant permission, ...

can lead to complications and errors.

virtual reality: modeling real-world objects

e.g., aircraft in air traffic control simulation,
atoms in a protein folding simulation, ...

queries: combinations of positions, orientations, speeds, etc.

updates: add, delete, change object states in many ways.

$\#q + \#u \longrightarrow \#q \times \#u$ worst case.

many others:

databases: especially for OLAP queries and updates.

network simulation: for performance analysis.

distributed systems: opening remote resources.

Achieving both clarity and efficiency

A powerful and systematic method for
incrementalization across object abstraction

1. allow "what" of each component to be specified clearly and modularly and implemented straightforwardly in an object-oriented language.

2. analyze queries and updates, across object abstraction, in the straightforward implementation.

3. transform into sophisticated and efficient "how" by incrementally maintaining the results of repeated expensive queries with respect to updates to their parameters.

Related work

incrementalization [many since 1960's, ideas centuries old]:
arithmetic operations, loops and arrays,
recursive functions and recursive data structures,
set and map operations, rules. not across object abstractions

optimization of OO programs [many since 1980's]:
method inlining,
method resolution,
other conventional optimizations. not incrementalization

analysis of OO programs [many since 1980's]:
much pointer analysis,
lacking performance analysis. not aimed at program clarity

Outline

motivation, overview, and related work

method, with a running example:

1. object abstraction: language w/sets, cost model, challenges

2. analysis: expensive queries, parameter updates, costs

3. transformation: incrementalization rules, composition

summary and discussion

applications and experiments:
   query optimization, role-based access control, ...

A wireless protocol example

a protocol keeps a set of signals and finds the set of signals whose strength is above a certain threshold:

component: Protocol
  data:
    signals: set of signals
    threshold: threshold for a signal to be strong
    ...
  operations:
    addSignal: add a given signal to the set of signals
    findStrongSignals: return the set of signals whose
        strength is above the threshold
    ...
component: Signal
  data:
    strength: strength of the signal
    ...
  operations:
    setStrength: set the strength to a given value
    getStrength: return the strength
    ...
...

1. Language and cost model

language: $\{v$ `in` $s \mid e\}$ is the set of v in s such that e holds on v.

`new set()`, s`.add(`v`)`, s`.remove(`v`)`, s`.any()`, s`.size()`, s`.contains(`v`)`

```
class Protocol
  signals: set(Signal)
  threshold: float
  ...
  addSignal(signal): signals.add(signal)
  findStrongSignals(): return {s in signals | s.getStrength() > threshold}
  ...
class Signal
  strength: float
  ...
  setStrength(v): strength = v
  getStrength(): return strength
  ...
...
```

like in set lang SETL,
query lang SQL,
specification lang Z,
modeling lang UML OCL,
scripting lang Python,
...

cost model: asymptotic running time. expensive: not $O(1)$.

for primitive and library op's: `size`: $O(|s|)$ or $O(1)$; others: $O(1)$.

Challenges of incrementalization across object abstraction

```
class Protocol
  signals: set(Signal)
  threshold: float
  addSignal(signal): signals.add(signal)
  findStrongSignals(): return {s in signals | s.getStrength() > threshold}

class Signal
  strength: float
  setStrength(v): strength = v
  getStrength(): return strength
```

expensive query: {`s in signals|s.getStrength() > threshold`}

where to store: a field of `Protocol`

where to update: `setStrength` in `Signal`? some method in `Protocol`?

how to update: a signal holds field of `Protocol`? holds a protocol?

many queries, many updates, interdependent: ...?

2. Analysis

expensive queries: non-O(1) basic op or compound comp.
(1) containing class and method,
(2) parameters read, $read(e)$, and
(3) cost and frequency.

primitive updates: write to var or field by assign or lib op.
(1) containing class and method,
(2) parameters written, $write(s)$, and
(3) cost and frequency.

costs and frequencies: can be absolute or relative.
extend automatic complexity analysis for $cost(op)$ and $freq(op)$.
can combine with user annotation & run-time monitoring.
easier for higher-level lang.: $cost(\{v \texttt{ in } s \mid e\}) = |s| \times cost(e)$

2. Analysis — determine expensive queries

{s in this.signals | s.getStrength() > this.threshold}

class: Protocol, method: findStrongSignals

parameters read: { this.signals,
this.signals.members,
{s.strength: s in this.signals},
this.threshold}

cost: $O(|\texttt{this.signals}|)$

$read(e)$:

$read(\{v \texttt{ in } s \mid e\}) = \{s, s.\texttt{members}\}$
$\cup \{\{p : v \texttt{ in } s\} : p \in read(e) \mid v \text{ appears in } p\}$
$\cup \{p : p \in read(e) \mid v \text{ appears not in } p\}$

2. Analysis — identify primitive updates

`this.signals.add(signal)`
- class: `Protocol`, method: `addSignal`
- parameters written: {`this.signals.members`}
- cost: O(1)

`this.strength = v`
- class: `Signal`, method: `setStrength`
- parameters written: {`this.strength`}
- cost: O(1)

$write(s)$: to variable or field by assignment or library operation

s is an update to query e: $\exists p \in write(s), q \in read(e) : p$ prefix of q
employing aliasing analysis.

3. Transformation—maintain single invariant

example:

inv $r = s.\texttt{size}()$	$O(\lvert s\rvert)$
at $s = \texttt{new set}()$	$O(1)$
do $r = 0$	$O(1)$
at $s.\texttt{add}(x)$	$O(1)$
do before	
`if not` $s.\texttt{contains}(x)$	$O(1)$
$r = r + 1$	
at $s.\texttt{remove}(x)$	$O(1)$
do before	
`if` $s.\texttt{contains}(x)$	$O(1)$
$r = r - 1$	

default: the query and all updates are in the same class.

in general:
- the query and all updates can be in different classes, or can all be in the same method of the same class.
- there can be conditions; there can be declarations.

3. Transformation — incrementalization rule

incrementalization rule:

$$
\begin{array}{ll}
\textbf{inv}\ r = \mathit{query} & \mathit{cost}_q \\
(\textbf{at}\ \mathit{update} & \mathit{cost}_u \\
\ \textbf{if}\ \ \mathit{condition} & \\
\ \textbf{de}\ (\mathit{variable}|\mathit{field})^* & \\
\quad (\textbf{in}\ C\ (\mathit{field}|\mathit{method})^+)^* & \\
\ \textbf{do before}\ \mathit{maint}_1 & \mathit{mcost}_u \\
\quad \textbf{after}\ \mathit{maint}_2)^* &
\end{array}
$$

1. declare variable r in m_q, if $C_u = C_q$, $m_u = m_q$ for all $update$'s; declare field r in C_q, otherwise.
2. replace each occurrence of query in C_q with r.
3. maintain $r = \mathit{query}$ incrementally: at each $update$, if $condition$ & if $\mathit{mcost}_u \leq \mathit{cost}_u$ or $\sum_u$ where $\mathit{mcost}_u > \mathit{cost}_u \mathit{mcost}_u \times \mathit{freq}_u < \mathit{cost}_q \times \mathit{freq}_q$
   - declare each $\mathit{variable}$ or field as for r in 1;
   - declare each field or method in class C;
   - insert maint_1 before update, and maint_2 after update.

3. Transformation — rule library

a rule for set comprehension: reuse

inv $r = \{v \texttt{ in } s \mid e\}$	$O(\lvert s\rvert \times cost(e))$
if $vars(e) \subseteq \{v, \texttt{this}\}$	
at $s = \texttt{new set()}$	$O(1)$
do $r = \texttt{new set()}$	$O(1)$
at $s.\texttt{add}(x)$	$O(1)$
do if $e[v \mapsto x]$	
$r.\texttt{add}(v)$	$O(cost(e))$
at $s.\texttt{remove}(x)$	$O(1)$
do if $e[v \mapsto x]$	
$r.\texttt{remove}(v)$	$O(cost(e))$

inv $r = \{v \texttt{ in } s \mid e\}$

$\cdots$

at *update* $O(cost(update))$

if s is a field of C_q, $type(s) = \texttt{set}(C_u)$, $C_u \neq C_q$,
$\{v.f : v \texttt{ in } s\} \in read_q$, and $write_u = \{\texttt{this}.f\}$

de in C_u

$c_q\texttt{s} : \texttt{set}(C_q)$

$\texttt{take}C_q(c_q) : c_q\texttt{s.add}(c_q)$

in C_q

$\texttt{update}C_u(x)$:

```
  if s.contains(x)
    if r.contains(x)
      if not e[v ↦ x]
        r.remove(x)
    else
      if e[v ↦ x]
        r.add(x)
```

at $s.\texttt{add}(x)$ $O(1)$

if $type(s) = \texttt{set}(C)$, $C \neq C_q$, and
there is an update to a field in C

do $x.\texttt{take}C_q(\texttt{this})$ $O(1)$

do after

$\texttt{for } c_q \texttt{ in } c_q\texttt{s}$

$c_q.\texttt{update}C_u(\texttt{this})$ $O(cost(e) \times |c_q\texttt{s}|)$

3. Transformation–maint. multiple invariants

independent queries: the parameters of each query do not depend on the results of other queries.

may apply all rules, simultaneously or one at a time in any order.

dependent queries: the parameters of some query depends on the results of other queries.

follow chains of dependencies among the queries.
this corresponds to the chain rule in calculus.

auxiliary optimizations:
auxiliary specialization, dead code elimination, fusion.

3. Transformation — example

```
  class Protocol
    signals: set(Signal)
    threshold: float
+   strongSignals: set(Signal)
    ...
    addSignal(signal): signals.add(signal)
+     signal.takeProtocol(this)
+     if signal.getStrength() > threshold
+       strongSignals.add(signal)
*   findStrongSignals(): return strongSignals
+   updateSignal(signal):
+     if signals.contains(signal)
+       if strongSignals.contains(signal)
+         if not signal.getStrength()>threshold
+           strongSingals.remove(signal)
+       else
+         if signal.getStrength()>threshold
+           strongSingals.add(signal)
    ...
```

```
  class Signal
    strength: float
+   protocols: set(Protocol)
    ...
+   takeProtocol(protocol):
+     protocols.add(protocol)
    setStrength(v):
      strength = v
+     for protocol in protocols
+       protocol.updateSignal(this)
    getStrength(): return strength
    ...
  ...
```

+	added lines
*	changed lines
	original lines

`findStrongSignal`: $O(|\texttt{signals}|) \rightarrow O(1)$. `setStrength`: $O(1) \rightarrow O(|\texttt{protocols}|)$.

Summary and discussion

clarity $\longrightarrow$ efficiency: incrementally maintain results of expensive queries with respect to parameter updates.

correctness: for concurrent prog too if maint. is atomic w/update.
cost model: can count constants, and consider space too.
usage: can be automatic, semi-automatic, or manual.

scales; suits incremental development: may use inheritance.
integrates OOP and AOP (Aspect-Oriented Programming).

rule derivation: generate rules for queries over sets and objects.
on-demand comp: move some maint. from updates to query.
concurrent comp: reduce synchronization of maint. w/updates.

Applications and experiments

applications: transforming clear and straightforward programs into sophisticated and efficient programs.

protocol example: 9 lines $\longrightarrow$ 17 added, 1 changed lines.
linear to constant query speedup, as analyzed.

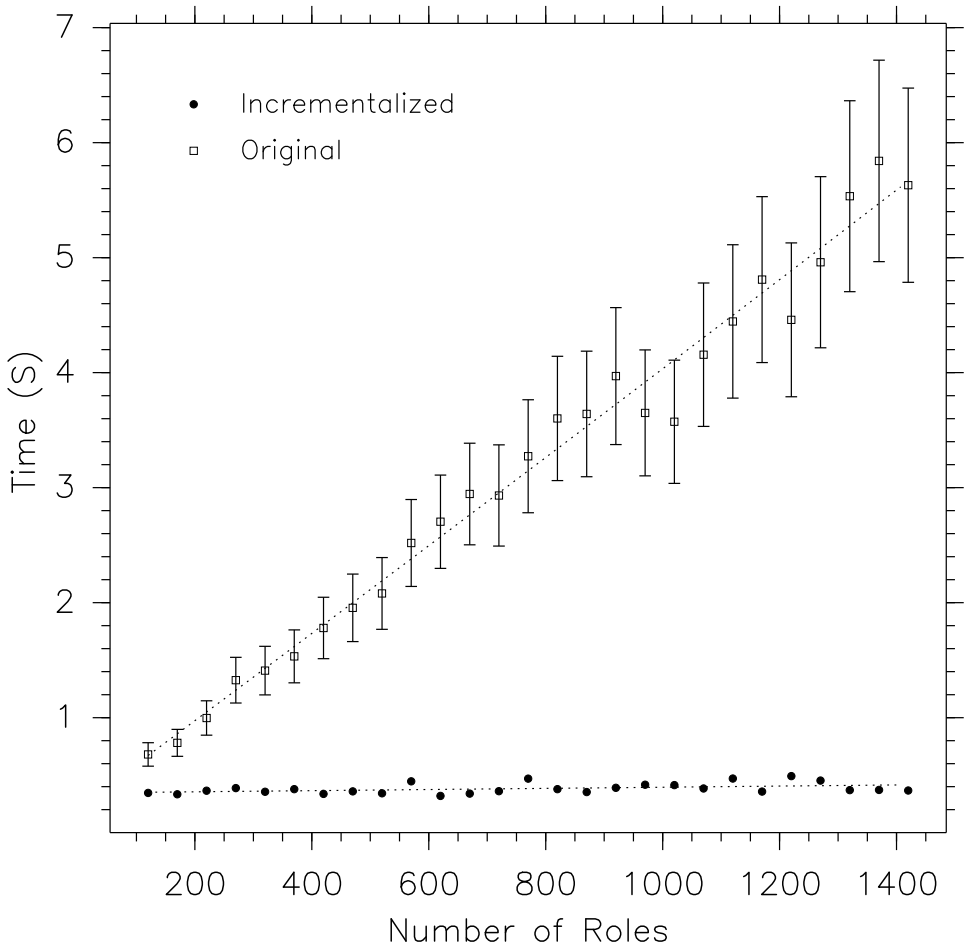
database join query: { `[x,y]: x in s, y in t | f(x)=g(y)` }
from quadratic to optimal, i.e., linear in input plus output.

graph reachability, ...

role-based access control (RBAC): ANSI standard, in Z.
sets of users, objects, operations, roles, and sessions, & over a dozen relations. several dozen ops. majority in core RBAC.

experiments: implemented analyses and transformations for Python in Python, applied them automatically, did measurements.

Applications and experiments — core RBAC



clarity $\longrightarrow$ efficiency:

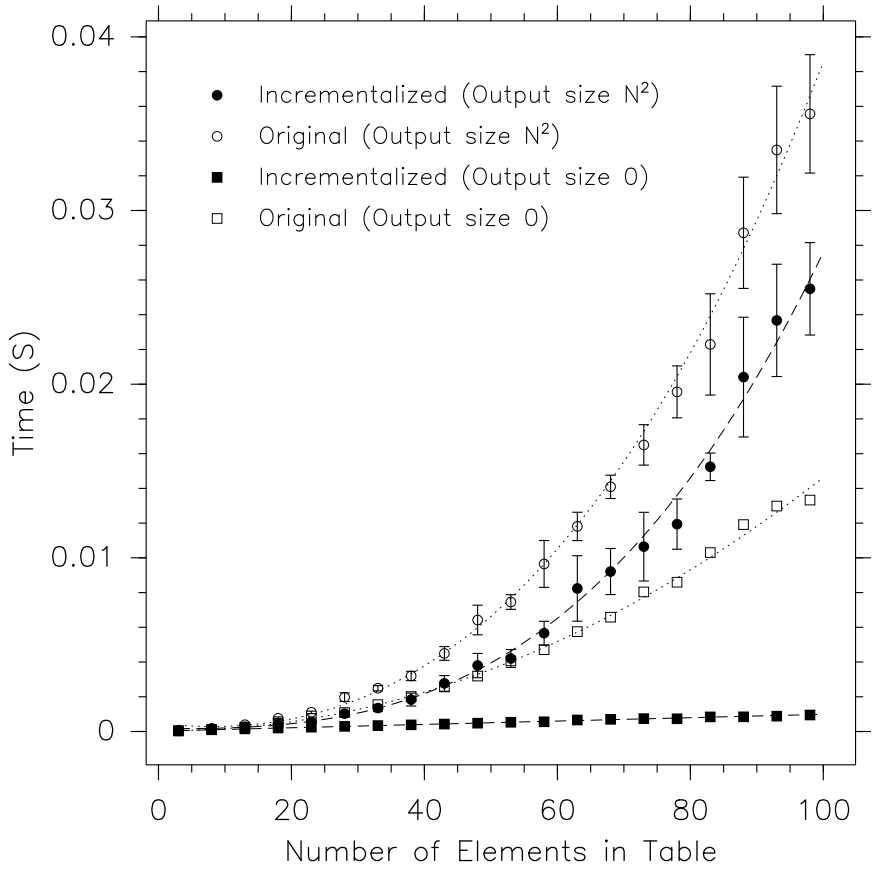
125 lines $\longrightarrow$ 610 lines.
for 7 expensive queries.
spread over updates.

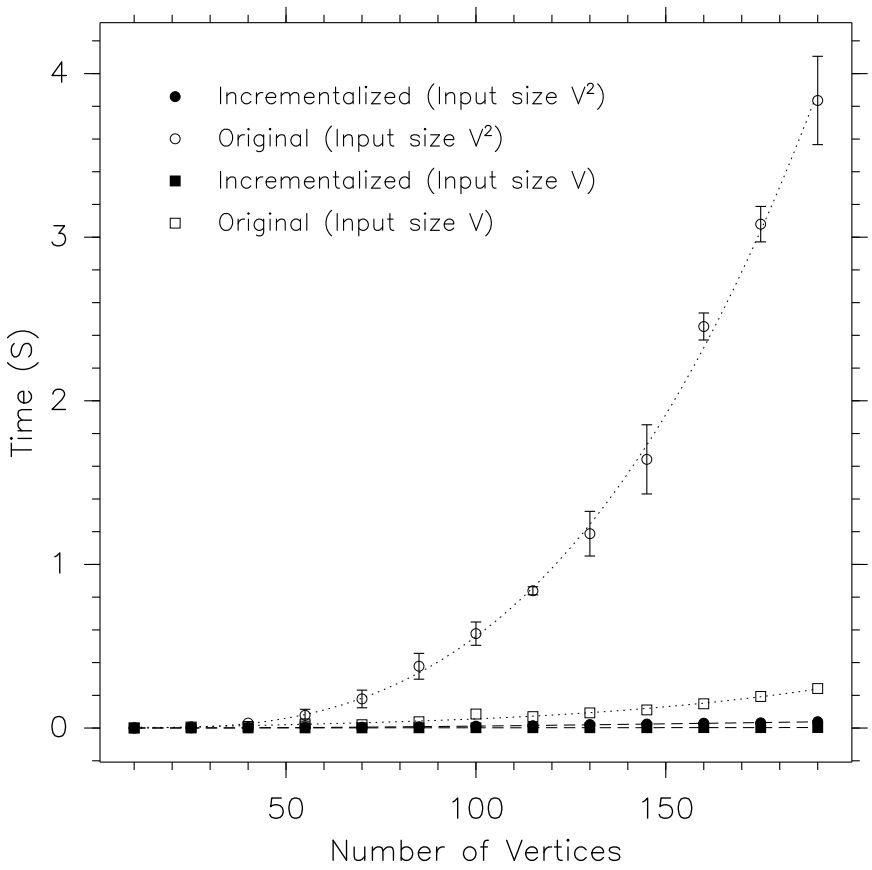
queries all become $O(1)$
but with tradeoffs, e.g.,
`CreateSession` R $\rightarrow$ r/s·p/r
`CheckAccess` R $\rightarrow 1$

1400 roles: `CheckAccess`'s
$>$ 5 sec $\longrightarrow$ $<$ 0.4 sec.

found a number of errors
and complications in the
ANSI standard.

Applications and experiments—join query and graph reachability





Conclusion

program development must resolve conflict between clarity and efficiency: need incrementalization across object abstraction.

a powerful and systematic method: analyze straightforward but inefficient implementation, and transform into efficient but sophisticated implementation, by incrementalizing expensive queries with respect to updates.

on-going projects: high-level languages (OO, sets, rules, regular path queries), analysis and transformations (methods and frameworks), implementations and experiments, security and other applications.

the dual problem:
given scattered incremental updates, determine the query—the invariant. essential for understanding legacy software.

discrete event simulation: model network packet transmission.

event list, implemented using `list` of C++ STL in gcc-2.95;

`size`→`empty`: 201→0.11s for 770s simulation of 5s 4M events