Incrementalization Across Object Abstraction*

Yanhong A. Liu[1] Scott D. Stoller[1] Michael Gorbovitski[1] Tom Rothamel[1] Yanni Ellen Liu[2]

[1] Computer Science Department, State University of New York at Stony Brook, Stony Brook, NY 11794

{liu,stoller,mickg,rothamel}@cs.sunysb.edu

[2] Department of Computer Science, University of Manitoba, Winnipeg, MB, R3T 2N2, Canada

yliu@cs.umanitoba.ca

ABSTRACT

Object abstraction supports the separation of what operations are provided by systems and components from how the operations are implemented, and is essential in enabling the construction of complex systems from components. Unfortunately, clear and modular implementations have poor performance when expensive query operations are repeated, while efficient implementations that incrementally maintain these query results are much more difficult to develop and to understand, because the code blows up significantly, and is no longer clear or modular.

This paper describes a powerful and systematic method that first allows the "what" of each component to be specified in a clear and modular fashion and implemented straightforwardly in an object-oriented language; then analyzes the queries and updates, across object abstraction, in the straightforward implementation; and finally derives the sophisticated and efficient "how" of each component by incrementally maintaining the results of repeated expensive queries with respect to updates to their parameters. Our implementation and experimental results for example applications in query optimization, role-based access control, etc. demonstrate the effectiveness and benefit of the method.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-Oriented Programming; D.2.1 [**Software Engineering**]: Requirements/ Specifications; D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.11 [**Software Engineering**]: Software Architectures; D.2.13 [**Software Engineering**]: Reusable Software; D.3.2 [**Programming Languages**]: Language classifications—*object-oriented languages, very high-level languages*; D.3.4 [**Programming Languages**]: Processors—*code generation, optimization*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*invariants*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*

General Terms

Design, Languages, Performance

Keywords

Abstraction, design, program transformation, incrementalization, invariants, program analysis, program optimization, object-oriented

*This work was supported in part by ONR under grants N00014-04-1-0722 and N00014-02-1-0363 and NSF under grants CCR-0306399, CCR-0204280, and CCR-0311512. Contact author: Y.A.Liu, liu@cs.sunysb.edu

OOPSLA'05, October 16–20, 2005, San Diego, California, USA.

1. INTRODUCTION

Object abstraction. One of the central concepts in computer science is abstraction [20, 40, 14, 26], which provides information encapsulation in systems and components by separating "what" from "how" in information processing. Abstraction is best supported as abstract data types [26], which are fundamental in modern high-level programming languages, including in particular object-oriented languages. An abstract data type provides an interface for a certain set of operations on a certain kind of data, i.e., the "what", shielding users from having to know how the data are represented and how the operations are implemented, i.e., the "how". This enables the construction of complex software systems by assembling software components.

What users do on data can be classified as, or decomposed into, two kinds of operations: queries and updates, where queries (which are sometimes called observations or views) compute results using data, and updates change data. For a simple example, consider the `LinkedList` class in Java 1.4. It has a query method `size` that returns the number of elements in the list, 11 update methods that add or remove elements, and several other query methods that return elements, their indices, a membership test result, etc.

How to implement the queries and updates can vary significantly. In a straightforward implementation, each operation does its respective query or update and is clear and modular. For example, in the `LinkedList` class, `size` would iterate over elements in the list to count them, and each of the 11 update methods would do exactly the specified addition or removal of elements. However, this can have poor

performance, because queries may be repeated and many are easily expensive. For example, `size` takes time linear in the number of elements in the list, and if it occurs in a loop, the overall performance blows up quickly. A sophisticated implementation can have good performance, by storing the results of expensive queries and maintaining them incrementally when the data are updated. For example, the `LinkedList` class may maintain the result of `size` in a field and simply return it when `size` is queried. However, this is less clear, less modular, and more error-prone, because each of the 11 update methods must also update this field appropriately.

Clearly, there is a conflict between clarity and efficiency, even for this simple `LinkedList` example. The situation becomes much worse for complex systems that may have many queries and updates, where queries may cross components and updates may also be spread in many components. It poses a serious challenge to consider all the complex dependencies and tradeoffs and to decide where and how to maintain what results, and the resulting code may become significantly more difficult to understand.

Conflict between clarity and efficiency in complex systems. The conflict between clarity and modularity, and thus software productivity and cost, on one side and program efficiency on the other side manifests itself widely in complex systems and components. We give two examples here; additional examples are discussed in Section 6.

Many simulations model real-world objects, such as aircraft in an air traffic control simulation or atoms in a protein folding simulation. Queries about the states of the system may combine the positions, orientations, speeds, and other attributes of the objects. At the same time, attributes of objects may change in many ways. If there are n kinds of queries and m kinds of changes, a straightforward but inefficient implementation would have $n+m$ clear and modular operations. A sophisticated implementation would maintain some of the results of the n kinds of queries and have each of the m kinds of changes also update each of the saved results, yielding $n \times m$ kinds of updates total in the worst case.

In database systems, the basic operations on data are clearly queries and updates, corresponding to `select` and `update` statements, respectively, in SQL. For efficiency, query results need to be saved as materialized views, and views need to be updated incrementally when the underlying data changes [9, 1]. While traditional OLTP (On-Line Transaction Processing) applications support relatively simple queries, OLAP (On-Line Analytical Processing) applications repeatedly answer much more expensive queries involving many large aggregates. This poses severe challenges in the implementation of OLAP applications.

Clear and modular implementations may have performance problems, while efficient implementations are often more complex, less clear, and more difficult to develop. The question is, then, does a method exist that allows us to achieve both clarity and efficiency?

This paper. This paper describes a powerful and systematic method for performing incrementalization across object abstraction that helps achieve both clarity and efficiency. The method first allows the "what" of each component to be specified in a clear and modular fashion and implemented straightforwardly in an object-oriented language. The method then analyzes the queries and updates, across object abstraction, in the straightforward implementation, and derives the sophisticated and efficient "how" of each component by incrementally maintaining the results of repeated expensive queries with respect to updates to their parameters.

There are currently few straightforward implementations of computer applications, but programmers would be significantly more productive if they could write straightforward implementations and rely on automated analysis and transformations to generate sophisticated and efficient implementations.

Contributions of this paper include:

- a powerful method for transforming straightforward expensive computations into efficient incremental computations across object abstraction, and thus turning clear and modular but inefficient implementations into sophisticated but efficient implementations;
- a method for analyzing the parameters read and updated by a computation in a language with object and set abstractions;
- a mechanism for defining incrementalization rules and for building a library of such rules, forming a knowledge base for powerful program transformations;
- a prototype implementation with applications in query optimization, role-based access control, etc. and with experimental results that demonstrate the benefit of the method.

Optimizations similar to incrementalization have been studied for various language features, e.g., [11, 22, 17, 38, 6, 45, 50, 31, 29, 30, 34, 28], but no systematic method generates incremental programs across abstraction in an object-oriented language. At the same time, many analyses and optimizations have been studied for object-oriented programs, e.g., [13, 5, 48, 15, 44, 3], but none of them achieves incrementalization. We build on and extend previous methods and present a systematic method for developing efficient object-oriented programs, a method that is scalable and well suited for incremental development.

The rest of the paper is organized as follows. Section 2 discusses the creation of object abstraction and challenges for incrementalization across object abstraction. Sections 3 and 4 describe the analyses and transformations, respectively, for incrementalization across object abstraction. Section 5 discusses related issues. Section 6 describes applications and experiments. Section 7 discusses related work and concludes.

2. OBJECT ABSTRACTION

We discuss the creation of object abstraction by specifying components from the users' perspective, constructing component operations in a clear and modular fashion using high-level operations, and making the cost model explicit. Such abstraction enables easier understanding, reuse, modification, enhancement, etc. of software systems and components. We then discuss the challenges of incrementalization across object abstraction.

Component specification. A software component captures data and operations on data that are of interest to the users.

As a running example, consider a wireless protocol that needs to keep, among other things, a set of signals and to find, among other things, the set of signals whose strength is above a certain threshold. This involves the following components.

```
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
    ...
...
```

One must first specify what the data and operations are from the users' perspective. The specification should be declarative, without considerations of how the data will be represented and how the operations will be implemented. The specification could be written in a high-level modeling language or specification language, such as Z [47], or in a very high-level programming language, as described below.

Clear and modular construction. Operations can be classified as, or decomposed into, queries and updates, where queries compute results using data but do not change data, and updates change data. For example, in the `Protocol` component, `findStrongSignals` is a query, and `addSignal` is an update; in the `Signal` component, `getStrength` is a query, and `setStrength` is an update. Each of the queries and updates can be constructed modularly and clearly in a straightforward fashion. Modularity here separates using data from changing data, and thus is clearer and less error-prone.

Figure 1 defines the language we use in this paper, to precisely present the analyses and transformations, although the principle underlying our method is general and applies to other languages as well. A program is a set of classes, each of which defines a set of fields and a set of methods. Types may be specified not only for fields but also for variables, method parameters, and return values, although we omit those types from the grammar. We generally omit types when they can be inferred from the program. Only side-effect-free methods may be invoked in expressions, and their bodies are always of the form `return` *expr*. We use indentation to indicate scoping. We make substantial use

$$
\begin{array}{rcl}
\mathit{prog} & ::= & \mathit{class}^* \\
\mathit{class} & ::= & \texttt{class}\ \mathit{classname} \\
 & & \quad (\mathit{fieldname}\colon \mathit{type})^* \\
 & & \quad (\mathit{methodname}(\mathit{varname}^*) : \mathit{stmt}^*)^* \\
\mathit{type} & ::= & \texttt{set}(\mathit{classname}) \mid \mathit{classname} \mid \texttt{int} \mid ... \\
\mathit{stmt} & ::= & \texttt{new}\ \mathit{classname}(\mathit{expr}^*) \\
 & & \mid \mathit{expr}.\mathit{methodname}(\mathit{expr}^*) \mid \texttt{return}\ \mathit{expr} \\
 & & \mid \mathit{expr}.\mathit{fieldname}{=}\mathit{expr} \\
 & & \mid \mathit{varname}{=}\mathit{expr} \mid \texttt{if}\ \mathit{expr}\ \mathit{stmt}\ \texttt{else}\ \mathit{stmt} \mid ... \\
\mathit{expr} & ::= & \{\mathit{expr} : (\mathit{varname}\ \texttt{in}\ \mathit{expr})^* \mid \mathit{expr}\} \\
 & & \mid \mathit{expr}.\mathit{methodname}(\mathit{expr}^*) \\
 & & \mid \mathit{expr}.\mathit{fieldname} \\
 & & \mid \mathit{varname} \mid \texttt{if}\ \mathit{expr}\ \mathit{expr}\ \texttt{else}\ \mathit{expr} \\
 & & \mid \mathit{expr}{+}\mathit{expr} \mid ...
\end{array}
$$

$\mathit{classname}, \mathit{fieldname}, \mathit{methodname}, \mathit{varname}$: identifiers

Figure 1: Language.

of sets, because they are well suited for expressing queries and updates at a very high level. Note the special type for sets and the special expression for set comprehension. For set comprehension $\{e : v_1\ \texttt{in}\ e_1, ..., v_k\ \texttt{in}\ e_k | e_b\}$, each variable v_i enumerates elements of the set value of expression e_i, and for each combination of values of v_1 through v_k, if the value of Boolean expression e_b is `true`, then the value of expression e forms an element of the resulting set. We abbreviate $\{v : v\ \texttt{in}\ e | e_b\}$ as $\{v\ \texttt{in}\ e | e_b\}$, and we omit $\mid e_b$ when e_b is `true`. We use the following notation for operations from a set component:

`new set`()	create and return an empty set
s.`add`(v)	add element v to s
s.`remove`(v)	remove element v from s
s.`contains`(v)	return `true` if v is in s and `false` o.w.
s.`any`()	return an arbitrary element in s
s.`size`()	return the number of elements in s

We use $\mathit{type}(e)$ to denote the type of expression e. As customary, we distinguish between object types and primitive types, i.e., non-object types, and use *Obj* to denote the set of object types. We treat set types as object types, except that we make the analysis more refined for sets. We use $\mathit{vars}(e)$ to denote the set of free variables in e. We use $e[x \mapsto e_1]$ to denote e but with each free occurrence of variable x in e replaced with e_1.

For the running example, the straightforward implementation in Figure 2 can be constructed. We use common syntactic conventions in examples, e.g., `signals` abbreviates `this.signals` in `addSignal`.

We make the following assumptions about straightforward, clear and modular programs. (1) Fields are initialized at object creation time. So it is safe to access the fields any time after an object is created. (2) Fields and sets are accessed only in the class where they are declared. So one only needs to look for updates to a field or set inside its class. (3) Library operations (whose code may be unavailable) come

```
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
  ...
...
```

Figure 2: Straightforward implementation.

with the information needed for the optimization (including parameters read and written, and cost), as described in Section 3.

Cost model. An essential feature of our method is the use of cost analysis to identify performance problems in straightforward implementations. As a basis for this, costs of primitive constructs in the language and operations from libraries, together with what cost is considered expensive in the application, should be specified explicitly. Frequencies of operations directly invoked by users of the application should also be provided if available.

In this paper, we use asymptotic running time as the main cost model (although other cost models could equally well be used), and we consider any operation whose cost is not $O(1)$ to be expensive. Expected costs of operations from the set component, assuming hashing is used in the implementation, are given in the table below, together with the parameters they read and write, respectively; operation s.`size`() may have a cost of $O(1)$ or $O(|s|)$ depending on how it is implemented in the set component.

	cost	*read*	*write*
`new set`()	$O(1)$		
s.`add`(v)	$O(1)$	s, s.`members`, v	s.`members`
s.`remove`(v)	$O(1)$	s, s.`members`, v	s.`members`
s.`contains`(v)	$O(1)$	s, s.`members`, v	
s.`any`()	$O(1)$	s, s.`members`	
s.`size`()	$O(1)$ or $O(\|s\|)$	s, s.`members`	

Our primary goal is to reduce the asymptotic running time of the incrementalized program. Of course, storing results of expensive queries takes extra space. Our secondary goal is to reduce space by maintaining only values useful for the optimization.

Incrementalization across object abstraction. First, expensive queries must be identified. For the program in Figure 2, it is easy to see that the set comprehension in `findStrongSignals` is an expensive query.

Next, we must examine where to store this query result, and where and how to update it. It is relatively easy to decide to store the result in a field of `Protocol`. For updates, there are more issues to consider. In particular, the update by `setStrength` in `Signal` may affect the query result. We want to incrementally maintain the stored query result as follows: if the strength of the signal is changed from above threshold to below, then the signal is removed from the query result; and if the strength is changed from below to above, then the signal is added. However, object abstraction makes this more difficult.

Should `setStrength` in `Signal` or some method in `Protocol` take care of the update? Clearly `setStrength` should initiate the update since it changes the signal strength. However, it can not directly access and update the query result in `Protocol` or access other data that is needed for the update but is not in `Signal`, whereas a method in `Protocol` can. Rather than giving `setStrength` access to those, a method can be defined in `Protocol` and called from `setStrength` to perform the update. How can a `Signal` object get a reference to a `Protocol` object to call the defined method? Note that all and only members of `signals` need to get such references. So a reference to the current `Protocol` object (i.e., `this`) can be passed to a `Signal` object when the `Signal` object is added to the `signals` field of the `Protocol` object. To do this, the `Signal` class must define a method for taking the reference to a `Protocol` object. Additionally, since a `Signal` object may be added to `signals` of multiple `Protocol` objects, a `Signal` object must maintain a set of references.

Finally, cost must be considered. Under what conditions will the transformations improve performance? In the straightforward implementation, each query takes $O(|$`signals`$|)$ time and each update takes $O(1)$ time; after the transformations, each query takes $O(1)$ time and each update takes $O(|$`protocols`$|)$ time, i.e., the number of instances of `Protocol`. So there is a tradeoff. In an application that has several or many signals but one or a few instances of `Protocol`, and where the query is performed at least as frequently as the signal strengths change, the transformed implementation is much more efficient.

The resulting implementation for the running example is shown in Figure 3, where `+` indicates an added line compared to Figure 2, and `*` indicates a modified line. Clearly, this implementation is significantly more complicated than the implementation in Figure 2, and automated support for such incrementalization is desired.

3. ANALYSIS

To perform incrementalization across abstraction, we need to determine expensive computations, identify updates to their parameters, and analyze costs and frequencies. The first two are described in two subsections below.

Costs and frequencies are used to decide when transformations improve performance. In general, they require separate analysis that is orthogonal to the main goal of this paper. They are made easier by the use of very high-level constructs like set comprehension. We use *cost*(`op`) to denote the cost of operation `op`, and *freq*(`op`) to denote the frequency of `op`. We extend existing automatic cost analysis [24, 35] to deal accurately with set comprehension, as follows:

$$\begin{aligned} &cost(\{e : v_1 \texttt{ in } s_1, ..., v_k \texttt{ in } s_k | e_b\}) = \\ &\quad (\textstyle\prod_{i=1..k} |s_i|) \times \max_{v_i \in s_i, i=1..k}(cost(e_b) + cost(e)) \end{aligned} \quad (1)$$

We assume that frequency information is given; otherwise, we may apply the transformation rules conservatively.

3.1 Determining expensive computations

There are two kinds of expensive computations: (1) a basic operation that is specified as expensive in the cost model,

```
  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
    ...
  ...
```

Figure 3: Incrementalized implementation.

such as a library operation that sorts a list, and (2) a compound computation that requires repeated operations, including a comprehension, an aggregation (e.g., the sum or minimum of a set of numbers), an iteration, and a recursion. For each expensive computation, three things must be determined: (1) containing class and method—the class and method where the computation appears, (2) parameters read—values that the computation depends on, and (3) cost—asymptotic running time of the computation. Note that expensive compound computations may have expensive subcomputations; we identify all expensive (sub)computations, and use the results of the subcomputations as parameters read by the supercomputation.

In the running example, in Figure 2, all basic operations used are inexpensive. The only expensive computation is the set comprehension:

```
{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(|this.signals|)
```

(2)

where the set of parameters read is the result of the analysis below.

The analysis computes the set $read(e)$ of parameters possibly read by an expensive computation e. To produce precise analysis results, we represent parameters using expressions of the form *par*:

$$par ::= ref | \{par : (varname \texttt{ in } ref)^*\}$$
$$ref ::= varname | ref.fieldname$$

where *ref* allows parts of objects to be captured, and *par* allows sets of parts of objects to be captured. For example, the third parameter read in (2) represents the `strength` field of elements of `this.signals`. To explicitly capture member objects of a set object s, we use s.`members`. For example, the second parameter read in (2) represents member objects of the set object, while the first parameter represents the reference to the set object. Finally, to appropriately select fields of objects, for an expression e of object type, the analysis also computes the set $valobj(e)$ of possible object references, each of form *ref*, for the value of e, excluding other intermediate values of e; it is a set since we allow conditional expressions in the language, and references from the branches are unioned.

Note that each parameter contains only one free variable, as explained below; this simplifies the analysis. A reference *ref* contains only one free variable, which appears on the left end. This follows immediately from the grammar. A parameter *par* contains one free variable (the grammar allows parameters with multiple free variables, such as {`x : x in s, y in t`}; the following argument implies that such parameters can be trivially simplified to contain one free variable). To see this, note that comprehension is of form $\{r_0 : v_1 \texttt{ in } r_1, ..., v_k \texttt{ in } r_k\}$ or $\{p : v_1 \texttt{ in } r_1, ..., v_k \texttt{ in } r_k\}$, where p is a comprehension, and each r_i is of form *ref* and has only one free variable. The only free variable in the first form is the free variable in r_1, because each v_i for $i = 1..k-1$ is the only free variable in r_{i+1}, and v_k is the only free variable in r_0. This also implies by induction that p has only one free variable, and the only free variable in the second form is also the free variable in r_1.

Figure 4 defines *read* and *valobj*, for queries that do not invoke recursive methods, and is explained below. Incrementalizing recursive functions has been studied previously [31, 29, 30]; integrating that with incrementalization of sets and objects is outside the scope of this paper. For simplicity, the analysis assumes that set-typed subexpressions are of form *ref*; this can always be achieved by introducing local variables or fields. For expressions of comprehension form, since they are considered expensive computations, local variables or fields will be introduced to store their values anyway.

For a set comprehension (read1), each s_i is of form *ref* and captures a reference to a set object, and s_i.`members` captures references to the member objects; they are included as parameters. Let ps be the set of parameters in the condition e_b or the return expression e. Comprehension notation is used to describe whole or parts (as indicated in parameter p from ps) of the elements of appropriate sets (as indicated by $v_{i_1} \texttt{ in } s_{i_1}, ..., v_{i_j} \texttt{ in } s_{i_j}$ that are selected from $v_1 \texttt{ in } s_1, ..., v_k \texttt{ in } s_k$). Parameters in ps whose free variable is not any of $v_1, ..., v_k$ are also included in the result.

For invocation of a user-defined method (read2), parameters of the method body e are first analyzed; let ps be the set of them. Then the analysis adds parameters of appropriate arguments (e_0 if `this` appears in a parameter in ps, and e_i if v_i appears in a parameter in ps), instantiations of each parameter p from ps with appropriate object references, using $valobj$, for arguments of object types (instantiating `this` to an object reference of e_0, and instantiating v_i to an object reference of e_i), and parameters in ps whose free variable is not any of $\texttt{this}, v_1, ..., v_k$. The definition of $valobj(e)$ is similar to $read(e)$, except for two differences. First, it is not defined for comprehensions or expressions with non-object

$read(\{e : v_1 \texttt{ in } s_1, ..., v_k \texttt{ in } s_k | e_b\})$ (read1)
$= \quad \cup_{i=1..k}\{s_i,\ s_i.\texttt{members}\}$
$\cup \{\{p : v_{i_1} \texttt{ in } s_{i_1}, ..., v_{i_j} \texttt{ in } s_{i_j}\} : p \in ps | vars(p) = \{v_{i_j}\},$
$vars(s_{i_j}) = \{v_{i_{j-1}}\}, ...,\ vars(s_{i_2}) = \{v_{i_1}\},$
$vars(s_{i_1}) \cap \{v_1, ..., v_k\} = \emptyset\}$
$\cup \{\, p \in ps | vars(p) \cap \{v_1, ..., v_k\} = \emptyset\}$
where $ps = read(e_b) \cup read(e)$

$read(e_0.m(e_1, ..., e_k))$ (read2)
where m is defined by $m(v_1, ..., v_k) : \texttt{return } e$
$= \quad \cup_{\{p \in ps | vars(p) = \{\texttt{this}\}\}} read(e_0)$
$\cup \cup_{\{p \in ps | vars(p) = \{v_i\}\}} read(e_i)$
$\cup \cup_{\{p \in ps | vars(p) = \{\texttt{this}\}\}} \{p[\texttt{this} \mapsto v] : v \in valobj(e_0)\}$
$\cup \cup_{\{p \in ps | vars(p) = \{v_i\},\ type(v_i) \in Obj\}} \{p[v_i \mapsto v] : v \in valobj(e_i)\}$
$\cup \{p \in ps | vars(p) \cap \{\texttt{this}, v_1, ..., v_k\} = \emptyset\}$
where $ps = read(e)$

$read(e_0.m(e_1, ..., e_k))$ (read3)
where m is in library, i.e.,$read(v_0.m(v_1, ..., v_k))$ is given
$=$ same as above except that
$read(e)$ is replaced with $read(v_0.m(v_1, ..., v_k))$

$read(e.f) = read(e) \cup \{v.f : v \in valobj(e)\}$ (read4)

$read(v) = \{v\}$ (read5)

$read(\texttt{if } e_1\ e_2 \texttt{ else } e_3)$ (read6)
$= read(e_1) \cup read(e_2) \cup read(e_3)$

$read(e_1 + e_2) = read(e_1) \cup read(e_2)$ (read7)

$valobj(e_0.m(e_1, ..., e_k))$
where m is defined by $m(v_1, ..., v_k) : \texttt{return } e$,
$= \quad \cup_{\{p \in ps | vars(p) = \{\texttt{this}\}\}} \{p[\texttt{this} \mapsto v] : v \in valobj(e_0)\}$
$\cup \cup_{\{p \in ps | vars(p) = \{v_i\},\ type(v_i) \in Obj\}} \{p[v_i \mapsto v] : v \in valobj(e_i)\}$
$\cup \{p \in ps | vars(p) \cap \{\texttt{this}, v_1, ..., v_k\} = \emptyset\}$
where $ps = valobj(e)$

$valobj(e_0.m(e_1, ..., e_k))$
where m is in library, i.e.,$valobj(v_0.m(v_1, ..., v_k))$ is given
$=$ same as above except that
$valobj(e)$ is replaced with $valobj(v_0.m(v_1, ..., v_k))$

$valobj(e.f) = \{v.f : v \in valobj(e)\}$

$valobj(v) = v$

$valobj(\texttt{if } e_1\ e_2 \texttt{ else } e_3) = valobj(e_2) \cup valobj(e_3)$

Figure 4: Analysis of parameters read by expensive computations.

types, because *valobj* is not called on them. Second, it only returns the values of e, not other values used when computing e.

For invocation of a method from a library (read3), the definition of the method is not given, but $read(v_0.m(v_1, ..., v_k))$ and $valobj(v_0.m(v_1, ..., v_k))$ are given. So, the analysis uses them instead of the results of analyzing the method body. The rest of the analysis is the same as for user-defined methods.

The parameters of a field reference $e.f$ (read4) include the parameters of e and the parameter $v.f$ for each object v that e may refer to.

Analysis rules for variable v (read5), conditional expression $\texttt{if } e_1\ e_2 \texttt{ else } e_3$ (read6), and expression $e_1 + e_2$ (read7) are straightforward. Analysis for other expressions of primitive types are similar as for $e_1 + e_2$.

The analysis is correct in that it returns all values that may be read by an expensive computation. It is precise in that it can capture parts of objects, including parts of elements of set objects, not just entire objects and sets. This analysis produces the parameters read shown in (2).

3.2 Identifying parameter updates

All updates to each parameter of each expensive computation must be identified. To minimize coordination effort and facilitate atomicity, we identify only updates by the primitives. There are two kinds of such primitive update operations: (1) an assignment statement that writes to a field or variable, and (2) a call to a library operation that writes to some fields. For each update operation u, three things need to be determined: (1) containing class and method, (2) parameters written—the fields or variables being updated, and (3) cost, as for expensive computations discussed above.

In the running example, in Figure 2, for expensive computation (2), there is an update to its parameter `this.signals.members`:

```
this.signals.add(signal)
```
class: `Protocol`, method: `addSignal` (3)
parameters written: {`this.signals.members`}
cost: O(1)

and there is an update to its parameter {`s.strength: s in this.signals`}:

```
this.strength = v
```
class: `Signal`, method: `setStrength` (4)
parameters written: {`this.strength`}
cost: O(1)

To determine whether a particular update is an update to a given parameter of a given query, a points-to or alias analysis is needed to detect whether the variable or object field being updated can be aliased with a variable or object field that the parameter represents. Any points-to or alias analysis can be used, provided it is modified to treat sets specially, following the approach for Java collections in [25]. In this approach, membership information is encoded as aliasing relationships involving the (imaginary) field `members` introduced above. In particular, for analyses based on points-to graphs, the fact that a node n represents an object that may be a member of a set represented by node n_s is encoded as the fact that $n_s.\texttt{members}$ may alias n, in other words, the points-to graph contains an edge $n_s \xrightarrow{\texttt{members}} n$; special transfer functions based on this encoding are used for set operations.

For a library operation $v_0.m(v_1, ..., v_k)$, information about which parts of its arguments are updated should be given. We assume that it is a set $write(v_0.m(v_1, \ldots, v_k))$ of parameters of the form *ref*. For example, for operations on sets, this information is given in the *write* column of the second table in Section 2.

An assignment to a variable, v, is an update to a read parameter p if p is v. An assignment to a field, $ref.f$, is an update to a read parameter p if p has the form $ref'.f$ or $\{ref'.f : v_1 \texttt{ in } s_1, \ldots, v_k \texttt{ in } s_k | e_b\}$, and *ref* may alias ref'. A call to a library operation, $e_0.m(e_1, \ldots, e_k)$, is an update to a read parameter $ref'.f$ if there exists $ref.f$ in $write(v_0.m(v_1, \ldots, v_k))$ such that $ref[v_0 \mapsto e_0, \ldots, v_k \mapsto e_k]$

may alias ref'. For simplicity, we assume here that $e_0, \ldots, e_k$ are variables; this can be achieved by introducing local variables.

Consider expensive computation (2) in the running example. Let $\texttt{this}_\texttt{a}$ and $\texttt{this}_\texttt{f}$ denote the `this` variables in methods `addSignal` and `findStrongSignals`, respectively. The update $\texttt{this}_\texttt{a}.\texttt{signals.add}(\texttt{signal})$ is an update to the read parameter $\texttt{this}_\texttt{f}.\texttt{signals.members}$ of (2), because $v_0.\texttt{members}$ is in $write(v_0.\texttt{add}(v_1))$, and $v_0[v_0 \mapsto \texttt{this}_a.\texttt{signals}]$ may alias $\texttt{this}_\texttt{f}.\texttt{signals}$, assuming that $\texttt{this}_\texttt{a}$ and $\texttt{this}_\texttt{f}$ may be aliased, i.e., that the elided part of the program in Figure 2 may call `addSignal` and `findStrongSignals` on the same instance of `Protocol`.

The analysis is correct in that all possible updates to a parameter of an expensive computation are identified. Its precision depends on the precision of the alias analysis. Imprecision may cause the incrementalization transformations to insert incremental maintenance code at updates where it is not needed. As part of the experiments described in Section 6, we found that Steensgaard's analysis [25] leads to some spurious updates, while Andersen's analysis [25] and Choi *et al.*'s analysis [10] are sufficiently precise to avoid spurious updates in the examples we considered.

4. TRANSFORMATION

We describe (1) transformations, expressed as incrementalization rules, that maintain a single invariant, i.e., the result of a single expensive query, under all updates to its parameters, (2) incrementalization rules for basic set operations and building a library of rules, and (3) handling multiple invariants.

4.1 Maintaining a single invariant

Consider an expensive query. We must decide (1) where to store the query result and (2) how to maintain the result at all updates to its parameters. Sometimes, maintaining a result requires maintaining additional results at the same time. Systematic methods for determining such additional results are outside the scope of this paper, but they have been studied for recursive functions [29, 30] and aggregate array computations [28], and we are currently developing methods for sets and objects, as summarized in Section 4.2. For these additional results, we must also answer the two questions above.

Part of the answer depends on whether the query and the updates are located across different methods and classes. There are three cases: (i) not all in the same class; (ii) all in the same class but not all in the same method; (iii) all in the same method of a class.

For storing the query result and additional results, in cases (i) and (ii), fresh fields are introduced in the class that contains the query; in case (iii), fresh variables in the same method as the query may be used. For ease of presentation, we always use r (for result) to denote the fresh field or variable for the query result.

For maintaining the query result and additional results at all updates, case (i) needs more coordination than cases (ii) and (iii). In all cases, the transformations are specified as *incrementalization rules* of the form:

$$\begin{array}{l}\textbf{inv}\ \ r = \mathit{query}\\ (\textbf{at}\ \ \mathit{update}\\ \ \textbf{if}\ \ \mathit{condition}\\ \ \textbf{de}\ \ (\mathit{variable}|\mathit{field})^*\\ \qquad (\textbf{in}\ C\ (\mathit{field}|\mathit{method})^+)^*\\ \ \textbf{do}\ \textbf{before}\ \mathit{maint}_1\\ \qquad \textbf{after}\ \mathit{maint}_2)^*\end{array} \tag{5}$$

where *query* and *update* are patterns for matching queries and updates, respectively, that are identified as in Section 3; *condition* is a test that involves information about *query* and *update*; *variable*, *field*, and *method* are declarations; C is the name of a class; and $maint_1$ and $maint_2$ are sequences of statements.

We denote the containing class and method, parameters read or written, and cost and frequency as C_q, m_q, $read_q$, $cost_q$, and $freq_q$, respectively, for *query*, and C_u, m_u, $write_u$, $cost_u$, and $freq_u$, respectively, for *update*. We use $mcost_u$ to denote the sum of the costs of $maint_1$ and $maint_2$ at *update*. An incrementalization rule applies if a query matches *query*, and every update to the parameters of the query matches at least one *update* and the corresponding condition holds, including the cost condition below; transformations for all matched updates must be applied together. Assuming the meta-variables in a rule, of form (5), have been instantiated for a specific query and for all updates to the query, the semantics of applying the rule is:

1. declare variable r in m_q, if $C_u = C_q$ and $m_u = m_q$ for all *update*'s; declare field r in C_q, otherwise;
2. replace each occurrence of *query* in C_q that has the same set of parameters with r, and
3. maintain $r = \mathit{query}$ incrementally as follows:
   at each *update*
   if *condition* holds and
   if $mcost_u \leq cost_u$ or
   $\sum_{u \text{ where } mcost_u > cost_u} mcost_u \times freq_u < cost_q \times freq_q$
   then
   - declare each *variable* in m_q, if $C_u = C_q$ and $m_u = m_q$ for all *update*s; declare *field* in C_q, otherwise;
   - declare each *field* or *method* in class C;
   - insert $maint_1$ before *update*, and $maint_2$ after *update*.

Condition and declaration clauses are optional. We assume that the variable, field, and method names in the declarations are not used in the given program; otherwise, fresh names can always be introduced for them. A declaration clause has no effect if the variable, field, or method to be added is already added by the same rule, since multiple *update*s may need to add the same declarations. For succinctness, a condition or declaration that is common for all *update*s in a rule may be hoisted above all *update*s. In $maint_1$ and $maint_2$, a fresh meta-variable denotes a fresh temporary local variable. One of $maint_1$ and $maint_2$ together with **before** or **after** may be omitted; when a **before** or **after** is omitted, the maintenance code may be inserted in either position.

A common condition is that $C_u = C_q$, so we include it by default if an **at**-clause does not explicitly specify other conditions on the containing classes. To facilitate cost consideration, we show costs of queries, updates, and maintenance to the side of each rule.

For example, the following rule maintains s.`size`() under s = `new set`(), s.`add`(x), and s.`remove`(x); by default, it applies when the query and all updates are in the same class:

```
inv  r = s.size()                          O(|s|)

 at  s = new set()                         O(1)
 do  r = 0                                 O(1)

 at  s.add(x)                              O(1)
 do before
     if not s.contains(x)                  O(1)
       r = r + 1

 at  s.remove(x)                           O(1)
 do before
     if s.contains(x)                      O(1)
       r = r - 1
```
(6)

For the running example, if there were expensive queries to return the size of the set of signals or the set of strong signals, then using the above rule, the sizes would be maintained incrementally at the updates.

4.2 Library of incrementalization rules

We describe important rules for maintaining set comprehension and aggregation and discuss libraries of incrementalization rules.

A rule for set comprehension. The rule in Figure 5 maintains a basic form of set comprehension under set initialization, element addition, element removal, and element modification. For element modification, the transformation for $C_q \neq C_u$ is shown; the transformation for $C_q = C_u$ is simpler and not presented.

The first three **at**-clauses are for updates in the same class as the query and are easy: for $r = \{v \texttt{ in } s|e\}$, when s is emptied, r is too; when an element x is added to s, the condition e is tested for x and, if it holds, x is added to r; similarly for element removal.

Consider the fourth **at**-clause, for element modification. The applicability condition is that s is a field of class C_q, elements of s are from class C_u, $C_u \neq C_q$, a field f of the elements is used in the query, and the update updates the field f of this instance of C_u. The transformations for incrementally maintaining r are as follows. First, in class C_u, a field c_q`s` is declared to keep a set of all C_q objects whose s field contains the C_u object, and a method `take`C_q is defined for adding a C_q object to the set c_q`s`; the last **at**-clause inserts code that calls `take`C_q when an element is added to s. Then, in class C_q, a method `update`C_u is defined for updating r given a C_u object x: if x is not in s, do nothing; otherwise, if x is in r but does not satisfy the condition e, remove x from r, and if x is not in r but satisfies e, add x to r. Finally, code is inserted after this element modification: for each object c_q in c_q`s`, `update`C_u is called on c_q with this C_u object as argument.

The new methods `take`C_q and `update`C_u and field c_q`s` work together as follows. Recall that the goal is to incrementally maintain in a C_q object the result of a query over a set of C_u objects when a C_u object is updated. First, when a C_u object is added to the set queried by the C_q object, method `take`C_q of the C_u object is called to add the C_q object to c_q`s`, the set of C_q objects that query over sets containing the C_u object. Then, when the C_u object is updated, it calls method `update`C_u for all C_q objects in c_q`s` to update the query results in those objects.

```
inv  r = {v in s|e}                        O(|s| × cost(e))
 if  vars(e) ⊆ {v, this}

 at  s = new set()                         O(1)
 do  r = new set()                         O(1)

 at  s.add(x)                              O(1)
 do  if e[v ↦ x]
       r.add(x)                            O(cost(e))

 at  s.remove(x)                           O(1)
 do  if e[v ↦ x]
       r.remove(x)                         O(cost(e))

 at  update                                O(cost(update))
 if  s is a field of C_q, type(s) = set(C_u), C_u ≠ C_q,
     {v.f : v in s} ∈ read_q, and write_u = {this.f}
 de  in C_u
     c_q s : set(C_q)     //c_q is C_q with lower case initial
     take C_q(c_q) : c_q s.add(c_q)
     in C_q
     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)
 do after
     for c_q in c_q s
       c_q.update C_u(this)                 O(cost(e) × |c_q s|)

 at  s.add(x)                              O(1)
 if  type(s) = set(C), C ≠ C_q, and
     there is an update to a field in C
 do  x.take C_q(this)                      O(1)
```

Figure 5: Rule for basic set comprehension.

For the running example, using the above rule, the incrementalized implementation in Figure 3 is obtained. The query (2) in `Protocol` can be incrementally maintained at updates (3) in `Protocol` and (4) in `Signal`. Incremental maintenance takes $O(|c_q\texttt{s}|)$ time at each update, which is $O(1)$ if there are only a few protocol instances, and the query time is reduced from $O(|\texttt{this.signals}|)$ to $O(1)$.

Rules for aggregations. An aggregation computes a quantity—such as size, sum, average, or minimum—over a set. We describe how these quantities may be maintained incrementally with respect to element addition or removal.

Size is easily maintained as in rule (6). Sum can be maintained similarly. The rule for average may declare and maintain additional results—namely *sum* and *count*—as well as the average. For a query that returns the minimum of a set, and updates that include element addition and deletion, incremental maintenance may use a black-red tree. It is easy to write an incrementalization rule that automatically declares, maintains, and uses a black-red tree when cost analysis shows that this is beneficial.

Library. A library of incrementalization rules can be built and reused. These rules would capture important algorithm

design and program development knowledge that is used repeatedly in constructing complex software systems. For example, the library may include rules for arithmetic operations, as used in strength reduction [11], rules for operations on bitwise data, as needed for hardware design [21], rules for operations on relations, as performed in databases, and rules for operations on state machines, for embedded applications.

When dealing with a library of rules, two important questions emerge. Where do the rules come from, and which rule do we apply when more than one rule can be used on an expensive query? We answer the first question here, and the second question in Section 4.3.

We have developed the rules in this paper in an ad-hoc manner, by analyzing the possible updates that can effect specific classes of expensive computations. This approach proves surprisingly effective in practice. We believe that one reason for this is that many of the expensive computations found in programs are not overly complex. These are computations that are currently being incrementalized by hand. By creating rules, we can automate this procedure and save the programmer a significant amount of effort.

As expensive computations get more complex, the situation gets more complicated. Developing, by hand, an incrementalization rule for a complex expensive computation is difficult. This is true both when developing a general rule that can be used with our system, and when trying to figure out how the expensive computation can be maintained by hand, as is the case today. While many computations can be incrementalized using a few rules, and new rules can be created to handle additional computations, we recognize that creating rules to handle complex computations may be a difficult and potentially error-prone process.

To address this, we are developing methods that compute incrementalization rules for entire classes of expensive computations. Such a method will take an expensive computation, and return a rule that can incrementalize that computation, and other similar computations. We have developed such a method for comprehensions. These comprehensions may contain any number of set iterations where a condition may involve multiple iteration variables, in contrast to the rule in Figure 5, which supports a single level of iteration; note that the rules in Figure 5 can be applied repeatedly to multiple iterations but each condition may involve only one iteration variable at a time. From the structure of the computation, our method derives additional values to maintain and code that executes when any field or set involved in the comprehension is updated.

We have also developed general methods that can be used to develop incrementalization rules. These methods allow for the incrementalization of recursive functions [29, 30] and aggregate array computations over loops [28]. Integrating these methods will increase the size of the rule library, and hence the set of programs that can be improved.

4.3 Multiple invariants and auxiliary optimizations

While an incrementalization rule specifies how to maintain the result of an individual query with respect to updates to its parameters, it is easy to see that the results of multiple independent queries, i.e., queries where the parameters of one query do not depend on the results of other queries, can be maintained simply by applying all the rules, whether simultaneously or one at a time in any order.

To maintain the results of multiple queries that are not independent, i.e., queries where the parameters of some query depends on the results of other queries, chains of dependencies among the queries must be followed. Dependencies are acyclic, because they are between two nested computations or two sequential computations. Incrementalization first maintains results of queries that do not depend on results of other queries, with respect to updates to its parameters, and then maintains queries that only depend on queries whose results have already been updated, with respect to updates to those results. This is like the chain rule in calculus [38].

In general, there may be multiple rules that apply to a query and all updates to its parameters. Since all the application conditions including cost considerations are specified explicitly in rules, systematic methods and automated tools should be developed to support the selection of the rule or rules that lead to the best performance. At the lowest level, this is known as data structure selection [43, 8, 42]. General methods based on incrementalization rules that work at all levels are open for further study.

Incrementalization may yield opportunities for additional optimizations, including specialization and dead code elimination. Specialization uses information in the context of a computation to simplify conditionals in the result of incrementalization. Dead code elimination removes code that is no longer needed because values computed by such code are now incrementally maintained. While generally these optimizations do not improve asymptotic complexity over incrementalization, they may reduce the running time and space usage by a constant factor or a constant amount and reduce the size of the resulting code.

5. DISCUSSION

Correctness, cost, and scalability. The transformations preserve semantics in the sense that the incrementalized program has the same behavior as the original program but improved performance. The correctness holds not only for sequential programs, but also for concurrent programs provided that each incremental maintenance is performed atomically with the corresponding update.

An essential characteristic of our method is the explicit analysis and use of cost and frequency information to ensure performance improvement. While this paper mainly uses asymptotic cost, the method is general and other cost models may be used, and time and space trade-off can be considered.

Since our method analyzes and transforms expensive queries one at a time, it scales well. As discussed in the next subsection, our method may also be applied to components or incomplete specifications and be used to support incremental development.

Incremental development. Our method is well suited for incremental development, because it preserves object abstraction in the original specification and allows easy handling of the addition and deletion of queries and updates, and thus also components, that occur in incremental development. A new transformed implementation may inherit from the previous transformed implementation.

If a query is added, our method simply incrementalizes the query with respect to all updates to the query parameters. If an update is added, our method considers all queries whose

parameters may be changed by this update: if a query can be maintained incrementally under this update using the same incrementalization rule used to maintain it under the existing updates, then we simply add maintenance code for the new update; otherwise, we try to incrementalize the query under all the updates including the new one, and use the result, which is the original implementation if no incrementalization rule applies, to replace the old incremental code. If a query is deleted, all maintenance code for it can be removed. If an update is deleted, we may check applicability of incrementalization rules that yield more specialized and more efficient maintenance code.

Note that when incremental developement and reuse of previous implementations are not of concern, we may generate more efficient code that cuts through abstractions.

OOP and AOP. Object-oriented programming (OOP) and aspect-oriented programming (AOP) [23] are widely studied paradigms for structuring abstractions.

In our method, the initial abstraction is modular and object-oriented, where classes straightforwardly encapsulate data and operations, while the incrementalization rules are aspect-oriented, where each rule is an aspect, ensuring that the invariant for the result of an expensive computation is maintained at all updates to its parameters. This natural integration of OOP and AOP is fundamental in resolving the conflict between clarity and efficiency. This invariant-driven view of AOP is also what drives the generative approach to AOP [46]. An important open problem is a good language for incrementalization rules that facilitates re-use of rules; existing languages for AOP are mainly for application programming.

Additional optimizations. The transformations in Section 4 maintain query results eagerly at every update to the query parameters. In some cases, on-demand computation (also called lazy computation) may be more efficient. That is, changes are collected at updates, and maintenance of the invariant is performed just before the query. Our method can be extended to introduce on-demand computation as appropriate based on cost analysis.

As mentioned above, our transformations preserve semantics also for concurrent programs with the stated atomicity condition. Ensuring atomicity requires synchronization. Using synchronization uniformly at all updates makes verification of correctness easier but can be costly. Analyses and transformations that eliminate unnecessary synchronization would be an essential part of a systematic method for developing correct and efficient concurrent programs.

6. APPLICATIONS AND EXPERIMENTS

To demonstrate the effectiveness of our analyses and transformation techniques, we applied them to a number of problems in various domains of interest. For each problem, we first wrote a straightforward program that solves it. We did not take efficiency into account when writing these programs, but instead strove to write a clear and understandable program that could handle the task.

We then used a system we developed to automatically apply incrementalization rules to the straightforward code. This system, named InvTS (Invariant-driven Transformation System), consists of over 5000 lines of Python. It takes as input a Python program, and produces as output an optimized version of that program after applying the incrementalization rules. For both the straightforward and incrementalized programs, we report the number of non-empty, non-comment, non-import lines in it.

We ran each program on input data to evaluate how its performance was improved by our method. As our problems are chosen from a wide variety of domains, there is no one way to create input data for all of them, so the input is described below with each problem. For each problem, we created a program that lets us vary the generated data in a way that is governed by one or more independent variables. In this way we can generate a number of inputs, for each of which we need to determine the running time of each program. For a particular input and program, we compute the running time by running the program repeatedly on the data until the standard deviation of the set of running times is less than 10 percent of the mean of the set of running times. For each problem, we graph the running times of the straightforward and incrementalized programs, and analyze how the running time has changed asymptotically.

Our current incrementalization rule library contains six rules, of which five are used in the examples given below. In addition to the rule in Figure 5, we developed other rules to handle expensive computations that arise in our test programs. We have rules that handle comprehensions with up to two iterations, and with up to two free variables. Other rules allow us to incrementalize comprehensions over the contents of dictionaries, rather than sets.

Our test programs are single-threaded, and were run under Windows XP SP2 on a dual-processor Athlon XP 2.8Ghz with 2 GB of memory, of which around 1.6 was free when running our programs. Our example programs, written in Python, were run under ActivePython 2.4 Build 244. This system was also used to run the incrementalizer, which took between 2 and 18 seconds per example.

All of the programs that our method has been applied to were written by us. Some may object to this, preferring that our method be applied to open source programs written by others. But widely used open-source programs were generally written with efficiency in mind, and the programmers generally incrementalized the repeated expensive computations by hand. Our method would not improve the performance of such programs. Instead, our method would allow programmers to write simpler, more readable, and more maintainable programs, and leave incrementalization to automated tools.

Protocol. Our first example is the running example used throughout this paper. When translated to Python, the straightforward version consists of 14 lines of code. The automatic incrementalization system detected the single expensive query in this code, and incrementalized it. The resulting Python code is 42 lines long. In this example, 13 lines made it through intact, 1 line was changed, and 29 lines of code were added. In a more complete application, it's possible that more updates to the signal strength and set of signals could be used. Since our method finds and generates code for each update, even more code would be generated.

To show the effectiveness of our system, we measured the performance of both programs. Figure 6 shows the results of a test in which we performed ten million `findStrongSignals` queries over a `Protocol` object containing a varying num-

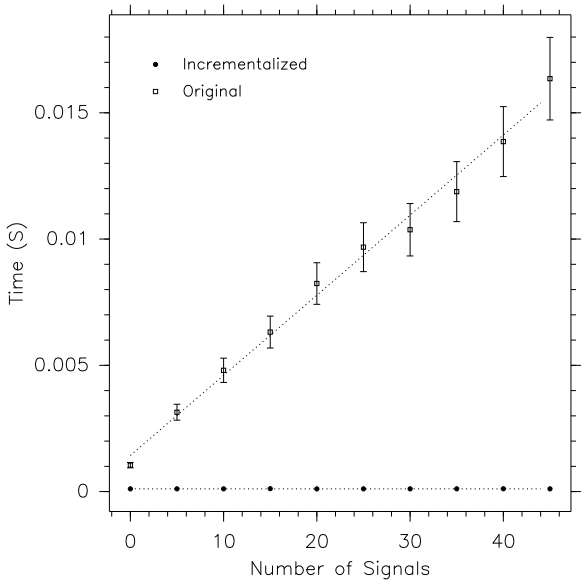

Figure 6: Running time of protocol. In all figures, the error bars denote a 95% confidence interval, while the lines are curves fit to the data points.

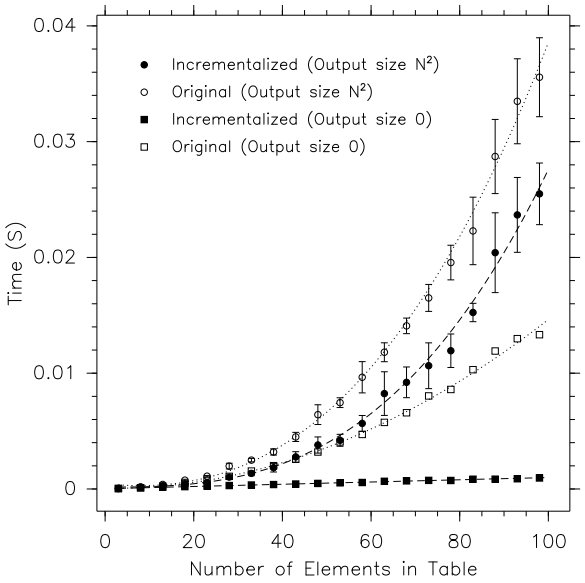

Figure 7: Running time of join.

ber of `Signal` objects. As one can see from Figure 6, the original straightforward program is linear in the number of signals present, while the incrementalized version of the program takes constant time, regardless of the number of signals found. This is as expected, and shows the benefits of incrementalization when the number of queries is much larger than the number of updates. Although we do not present it here, we did compare the running time of the program with a varying number of queries. As expected, the running time scales linearly with the number of queries performed, with the non-incrementalized version having an increasing slope as the number of `Signal` objects increases.

Join. Our second example is a join operation, which can be written as a comprehension of the form:

```
{[x,y]: x in s, y in t | f(x)=g(y)}
```

Converting this to Python and adding some support code gives a program of 10 lines in length, corresponding to the following high-level program:

```
result = new set()
for x in s
  for y in {y in t | f(x)=g(y)}
    result.add([x,y])
```

This program contains a single expensive computation, the comprehension `{y in t | f(x)=g(y)}`. When incrementalized over updates to the `s` and `t` sets, the program expands to 24 lines in length.

We created two series of test data to evaluate the performance of the straightforward and incrementalized versions of join. In both series, the size of the two input sets was given as the independent variable `N`. The series differ in the size of the output. One series produces output of size N^2, while the other produces no output at all, as would be the case with fully disjoint input. These two series let us explore the full limits of possible running times.

We ran both programs on both series of inputs. Figure 7 shows the running times. The straightforward program is always quadratic in running time, while the incrementalized program is quadratic or linear, depending on the size of the output. Thus, starting with a quadratic specification of join, we automatically obtained an incrementalized implementation that runs in time proportional to the size of the input and output; this is asymptotically optimal [49, 16].

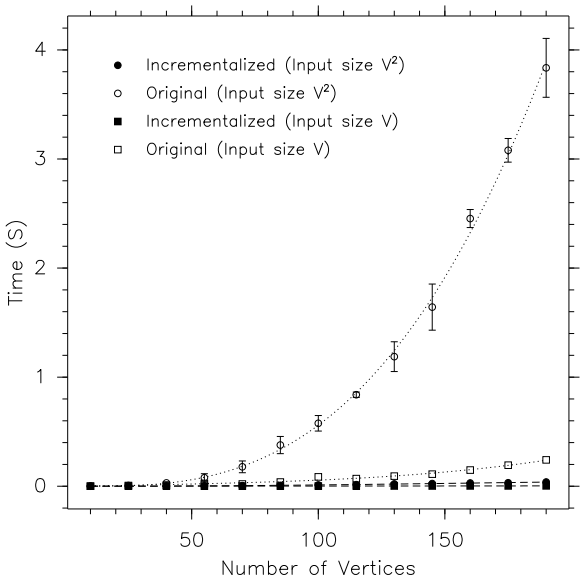

Figure 8: Running time of graph reachability.

Graph reachability. Our next example is graph reachability. In this problem, we are given a set `edges` containing pairs of vertices representing edges in a directed graph, and a starting vertex `source`. We compute all vertices reachable

from the starting vertex. A straightforward implementation takes 15 lines of Python code, corresponding to the following high-level program:

```
reach = new set()
reach.add(source)
while exists y in {y: x in reach, [x,y] in edges
                    | y not in reach}
  reach.add(y)
```

where `exists x in s` returns true if `s` not empty and binds `x` to an arbitrary element of `s`, and returns false otherwise. Our tool finds two expensive computations that can be incrementalized, and expands the program to 32 lines in length.

To evaluate the benefit of the optimization, we ran our programs on test data consisting of connected graphs with numbers `E` of edges that increase linearly and quadratically with the number of vertices, `V`. Figure 8 shows the running times of the programs on various input sets. The straightforward program took $O(\texttt{VE})$ time to compute all reachable vertices, which may be cubic or quadratic in the number of vertices, depending on the number of edges in the graph. The incrementalized version took $O(\texttt{E})$ time, saving time proportional to the number of vertices, making the running times quadratic and linear, respectively. In all cases, the incrementalized program ran faster than the straightforward one.

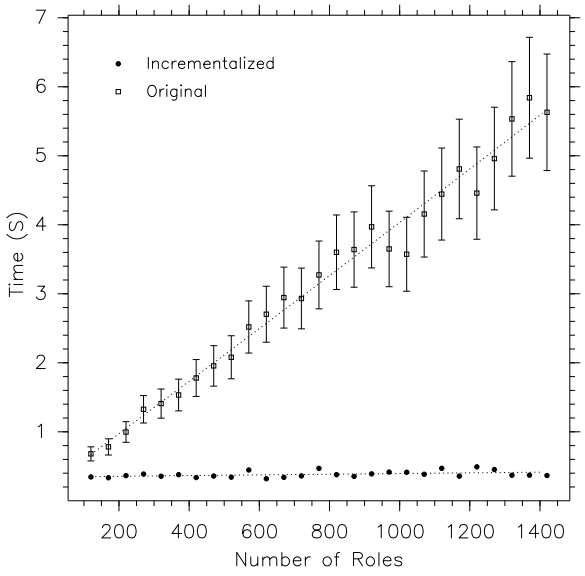

Figure 9: Running time of role-based access control.

Role-based access control. Our final example is our most realistic and complex one, and the one that shows the greatest improvements in running time. Starting from the specification in [2], we developed a straightforward high-level Python specification for Core Role-Based Access Control (Core RBAC). This high-level specification, consisting of 125 lines of interesting code, contains seven expensive queries that could be incrementally maintained. When incrementalized, the code more than quadrupled in size to 610 lines of lower-level Python code.

We measured the performance on input data that simulates a user access pattern. This pattern consists of a session creation, ten random access checks, and a session deletion operation. Each run consists of 100,000 repeats of this pattern, applied to a database containing a varying number of roles. Figure 9 shows the results. The straightforward code takes time proportional to the number of roles in the database, while the incrementalized program takes a constant amount of time. This improvement in asymptotic behavior leads to a large practical speedup. With a database containing 1,400 roles, the straightforward version takes over 5 seconds, while its incrementalized counterpart takes less than 0.4 seconds.

Now, realistic implementations of Core RBAC do not take seconds to process a million access checks. These implementations are incrementalized by their creators. Our method ensures that the incrementalization process is done explicitly and correctly, and saves the programmer time by reducing the amount of code that must be written by a factor of four. Furthermore, the code being saved is often the most tedious and complex in the program. In the RBAC example, over four hundred lines of code are added to incrementalize a mere seven expensive computations.

7. RELATED WORK AND CONCLUSION

There is a vast amount of research on object abstraction in specification and implementation of software. What has been lacking in this area is the explicit specification and use of cost models. Adding performance information in extended interfaces was proposed [12, 4], but no systematic optimization was developed that uses such information.

There is a large body of work on formal specification and transformation for program development (e.g., [17, 32, 41, 33]), including in particular methodologies for strengthening and maintaining invariants (e.g., [18]). Most of these methods do not consider cost explicitly, and most of them are not systematic or automatable. The exceptions are a number of methods based explicitly on making computation incremental (e.g., [38, 36, 50, 45, 19, 34, 27, 28]), but none of them addresses the development of object-oriented programs.

In particular, the finite differencing method for incrementally computing set expressions [38, 36] also exploits the chain rule and is particularly systematic. Together with data structure selection for implementing sets [37, 8], it has been used successfully in developing new, efficient algorithms and implementations for complex analysis problems (e.g., [39, 7]). Nevertheless, it only transforms straight-line code that has no procedural abstraction, let alone object abstraction.

Our incrementalization across object abstraction unifies and extends previous incrementalization methods into a systematic method for developing efficient object-oriented programs. Strength reduction, finite differencing, and other transformations for incremental computation can all be expressed as incrementalization rules. The method is scalable, since it transforms one query at a time, and is well suited for incremental development.

There is a large amount of work on transformation and optimization of object-oriented programs, e.g., [13, 5, 48, 15, 44, 3]. None of it performs incremental maintenance of invariants that is essential for transforming clear and modular but inefficient implementations into efficient but sophisticated implementations.

In conclusion, a systematic method for incrementalization across object abstraction is important in resolving the con-

flict between clarity and efficiency. Future work is needed on improved techniques for analyzing dependencies and analyzing costs and tradeoffs, suitable languages for specifying incrementalization rules, and further optimizations for on-demand and concurrent computations. Finally, the dual or reverse problem of incrementalization—given scattered incremental updates in a sophisticated implementation, determine the high-level query, i.e., the invariant—is very important for understanding legacy software.

8. REFERENCES

[1] M. A. Ali, A. A. A. Fernandes, and N. W. Paton. Movie: an incremental maintenance system for materialized object views. *Data Knowl. Eng.*, 47(2):131–166, 2003.

[2] American National Standards Institute, Inc. Role-based access control. ANSI INCITS 359-2004. Approved Feb. 19, 2004. http://csrc.nist.gov/rbac/.

[3] C. S. Ananian and M. Rinard. Data size optimizations for Java programs. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*, pages 59–68. ACM Press, 2003.

[4] F. Bachmann et al. Volume II: Technical concepts of component-based software engineering, 2nd edition. Technical Report CMU/SEI-2000-TR-008, Software Engineering Institute, 2000.

[5] D. F. Bacon and P. F. Sweeney. Fast static analysis of c++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 324–341. ACM Press, 1996.

[6] R. S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Trans. Program. Lang. Syst.*, 6(4):487–504, Oct. 1984.

[7] B. Bloom and R. Paige. Transformational design and implementation of a new efficient solution to the ready simulation problem. *Science of Computer Programming*, 24(3):189–220, 1995.

[8] J. Cai, P. Facon, F. Henglein, R. Paige, and E. Schonberg. Type analysis and data structure selection. In Möller [33], pages 126–164.

[9] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 577–589. Morgan Kaufmann Publishers Inc., 1991.

[10] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–245, Charleston, South Carolina, 1993.

[11] J. Cocke and K. Kennedy. An algorithm for reduction of operator strength. *Commun. ACM*, 20(11):850–856, Nov. 1977.

[12] D. Cohen, N. Goldman, and K. Narayanaswamy. Adding performance information to ADT interface. In *Proceedings of the Workshop on Interface Definition Languages*, pages 84–93, 1994. Published as SIGPLAN Notices, 29(8).

[13] J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers. Vortex: an optimizing compiler for object-oriented languages. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 83–100. ACM Press, 1996.

[14] E. W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, 1972.

[15] A. Diwan, K. S. McKinley, and J. E. B. Moss. Using types to analyze and optimize object-oriented programs. *ACM Trans. Program. Lang. Syst.*, 23(1):30–72, 2001.

[16] D. Goyal and R. Paige. The formal reconstruction and improvement of the linear time fragment of willard's relational calculus subset. In R. Bird and L. Meertens, editors, *Algorithmic Languages and Calculi*, pages 382–414. Chapman & Hall, London, U.K., 1997.

[17] D. Gries. *The Science of Programming*. Springer-Verlag, New York, 1981.

[18] D. Gries. A note on a standard strategy for developing loop invariants and loops. *Science of Computer Programming*, 2:207–214, 1984.

[19] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 157–166, May 1993.

[20] C. A. R. Hoare. Proof of correctness of data representation. *Acta Informatica*, 1:271–281, 1972.

[21] S. D. Johnson, Y. A. Liu, and Y. Zhang. A systematic incrementalization technique and its application to hardware design. *International Journal on Software Tools for Technology Transfer*, 4(2):211–223, 2003.

[22] S. Katz. Program optimization using invariants. *IEEE Trans. Softw. Eng.*, SE-4(5):378–389, Nov. 1978.

[23] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings of the 11th Europeen Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[24] D. Le Métayer. Ace: An automatic complexity evaluator. *ACM Trans. Program. Lang. Syst.*, 10(2):248–266, Apr. 1988.

[25] D. Liang, M. Pennings, and M. J. Harrold. Extending and evaluating flow-insenstitive and context-insensitive points-to analyses for Java. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 73–79. ACM Press, 2001.

[26] B. Liskov and S. Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, pages 50–59, Apr. 1974.

[27] Y. A. Liu and S. D. Stoller. From Datalog rules to efficient programs with time and space guarantees. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 172–183, Aug. 2003.

[28] Y. A. Liu, S. D. Stoller, N. Li, and T. Rothamel. Optimizing aggregate array computations in loops. *ACM Trans. Program. Lang. Syst.*, 27(1):91–125, Jan. 2005.

[29] Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Static caching for incremental computation. *ACM Trans. Program. Lang. Syst.*, 20(3):546–585, May 1998.

[30] Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Strengthening invariants for efficient computation. *Science of Computer Programming*, 41(2):139–172, Oct. 2001. An earlier version appeared in *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*, 1996.

[31] Y. A. Liu and T. Teitelbaum. Systematic derivation of incremental programs. *Science of Computer Programming*, 24(1):1–39, Feb. 1995.

[32] L. G. L. T. Meertens, editor. *Program Specification and Transformation*. North-Holland, Amsterdam, 1987. Proceedings of the IFIP TC2/WG 2.1 Working Conference on Program Specification and Transformation, Bad Tölz, FRG, April 1986.

[33] B. Möller, editor. *Constructing Programs from Specifications*. North-Holland, Amsterdam, 1991.

[34] H. Nakamura. Incremental computation of complex object queries. In *Proceedings of the 16th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 156–165. ACM Press, 2001.

[35] F. Nielson, H. R. Nielson, and H. Seidl. Automatic complexity analysis. In *Proceedings of the 11th European Symposium on Programming*, volume 2305 of *LNCS*, pages 243–261. Springer-Verlag, Berlin, 2002.

[36] R. Paige. Programming with invariants. *IEEE Software*, 3(1):56–69, Jan. 1986.

[37] R. Paige. Real-time simulation of a set machine on a RAM. In *Computing and Information, Vol. II*, pages 69–73. Canadian Scholars Press, 1989. Proceedings of ICCI '89: The International Conference on Computing and Information, Toronto, Canada, May 23-27, 1989.

[38] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Trans. Program. Lang. Syst.*, 4(3):402–454, July 1982.

[39] R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, Dec. 1987.

[40] D. L. Parnas. A technique for software module specification with examples. *Commun. ACM*, 15(5):330–336, May 1972.

[41] H. A. Partsch. *Specification and Transformation of Programs—A Formal Approach to Software Development*. Springer-Verlag, Berlin, 1990.

[42] T. Rothamel. On automatic data structure selection. Technical Report DAR 04-13, Computer Science Department, SUNY Stony Brook, May 2004.

[43] E. Schonberg, J. Schwartz, and M. Sharir. An automatic technique for the selection of data representations in SETL programs. *ACM Trans. Program. Lang. Syst.*, 3(2):126–143, Apr. 1981.

[44] U. P. Schultz, J. L. Lawall, and C. Consel. Automatic program specialization for java. *ACM Trans. Program. Lang. Syst.*, 25(4):452–499, 2003.

[45] D. R. Smith. KIDS: A semiautomatic program development system. *IEEE Trans. Softw. Eng.*, 16(9):1024–1043, Sept. 1990.

[46] D. R. Smith. A generative approach to aspect-oriented programming. In *Proceedings of the 3rd International Conference on Generative Programming and Component Engineering*, volume 3286 of *LNCS*, pages 39–54, Vancouver, Canada, Oct. 2004. Springer-Verlag.

[47] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 2 edition, 1992.

[48] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 187–206. ACM Press, 1999.

[49] D. E. Willard. Quasilinear algorithms for processing relational calculus expressions (preliminary report). In *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 243–257. ACM Press, 1990.

[50] D. M. Yellin and R. E. Strom. INC: A language for incremental computations. *ACM Trans. Program. Lang. Syst.*, 13(2):211–236, Apr. 1991.