Composing Transformations for Instrumentation and Optimization *

Michael Gorbovitski Yanhong A. Liu Scott D. Stoller Tom Rothamel

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

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

Abstract

When transforming programs for complex instrumentation and optimization, it is essential to understand the effect of the transformations, to best optimize the transformed programs, and to speedup the transformation process. This paper describes a powerful method for composing transformation rules to achieve these goals.

We specify the transformations declaratively as instrumentation rules and invariant rules, the latter for transforming complex queries in instrumentation and in programs into efficient incremental computations. Our method automatically composes the transformation rules and optimizes the composed rules before applying the optimized composed rules. The method allows (1) the effect of transformations to be accumulated in composed rules and thus easy to see, (2) the replacements in composed rules to be optimized without the difficulty of achieving the optimization on large transformed programs, and (3) the transformation process to be sped up by applying a composed rule in one pass of program analyses and transformations instead of applying the original rules in multiple passes.

We have implemented the method for Python. We successfully used it for instrumentation, in ranking peers in BitTorrent; and for optimization of complex queries, in the instrumentation of BitTorrent, in evaluating connections of network hosts using NetFlow, and in generating efficient implementations of Constrained RBAC.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automatic Programming—Program transformation; D.3.4 [*Programming Languages*]: Processors—Optimization; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Invariants

General Terms Design, Languages, Performance

1. Introduction

Program instrumentation and optimization are key tasks for program understanding and improvement. Instrumentation adds code to monitor program behavior at runtime, for both correctness and performance reasons. Optimization replaces inefficient code with efficient code and must preserve program semantics. For better program understanding, instrumentation must support complex queries of program behaviors. For better program improvement, optimization must transform complex queries, in instrumentation in particular and in programs in general, into efficient incremental computations with respect to updates to the values that the queries depend on; such optimizations are called *incrementalization*.

Program instrumentation and optimization can be expressed using transformation rules, which are then applied to programs. This allows complex instrumentations to be easily turned on or off, and complex optimizations to be reused for different applications, while at the same time allowing the original programs to be easier to understand. However, three problems must be addressed to fully support this approach: (1) the effect of applying a combination of rules can be hard to understand, (2) the efficiency of the resulting programs might not be the best from a combination of separate rules, and (3) the application of many rules can be too slow due to necessary complex program analysis being repeated.

This paper describes a powerful method for composing transformation rules to address these three problems. We specify the transformations declaratively as instrumentation rules and invariant rules, the latter for transforming complex queries in instrumentation and in programs into efficient incremental computations. Our method automatically composes the transformation rules and optimizes the composed rules before applying the optimized composed rules.

1. The method starts with complex queries needed in instrumentation or in programs, decomposes them into subqueries for which individual transformation rules may apply, and combines individual rules by repeatedly matching the replacement-pattern parts of a rule against the given-pattern parts of another rule. This allows the effect of transformations to be accumulated in composed rules and thus easier to understand through the rules.
2. The method then optimizes the replacement-pattern parts of composed rules through algebraic simplifications of composed computations and precise elimination of dead computations. This allows the replacement-pattern parts in composed rules to always be optimized without the difficulty of achieving similar optimizations on large transformed programs. This also allows the effect of transformations to be even easier to see.
3. The method finally applies the optimized composed rules to the program, employing powerful control flow, data type, and alias analysis to ensure that program semantics is preserved. This allows the transformation process to be sped up by applying a composed rule in one pass of program analyses and transformations in place of separate rules in multiple passes.

Our method ensures that applying a composed rule or an optimized composed rule yields programs that have the same semantics as programs obtained by applying individual rules.

We have implemented the method for transforming Python programs. Our implementation handles the entire Python 2.5 language. We have successfully used our system for instrumentation in ranking peers in BitTorrent—a peer-to-peer distributed file sharing program. We also successfully used the system for optimization of

* This work was supported in part by ONR under grants N000140910651 and N000140710928; NSF under grants CCF-0964196, CNS-0831298, CCF-0613913, and CNS-0509230; and AFOSR under grant FA0550-09-1-0481.

PEPM'12, January 23–24, 2012, Philadelphia, PA, USA.

complex queries in the instrumentation of BitTorrent, in evaluating connections of network hosts using NetFlow—a Cisco network protocol for collecting IP traffic information, and in generating efficient implementations from formal specifications for Constrained RBAC—advanced components in the ANSI standard for Role-Based Access Control (RBAC). We present experimental results that demonstrate the effectiveness and benefits of the method.

Much work has been done on program transformations and many related topics, including invariant rules and incrementalization, aspect-oriented programming, and many program transformation systems and applications, as discussed in Section 7. However, no previous work has studied composition of invariant rules and achieved the kind of optimizations possible using our method.

2. Transformation language

We slightly extend invariant rules from previous work [Liu et al. 2009] to add instrumentation rules. Invariant rules are designed to support the fundamental concept of maintaining invariants in programs. Instrumentation rules are designed to facilitate preserving semantics of programs when desired.

Invariant rules. Invariant rules are designed for incrementalization, i.e., optimizing expensive queries in programs by storing the query result and incrementally maintaining the result when the values that the query depends on are updated. This maintains the invariant that the value of the result variable always equals the result of the query. By queries, we mean computations of results using given values.

For example, the invariant rule in Figure 1 maintains the invariant that the value of `$r` always equals the result of `$s.len()`, the size of set `$s`, under three kinds of updates to `$s`:

1. when `$s` is assigned a new empty set, `$r` is assigned 0;
2. when adding an element `$x` to `$s`, `$r` is incremented by 1 if `$x` is not in `$s`; and
3. when removing an element `$x` from `$s`, `$r` is decremented by 1 if `$x` is in `$s`.

If all possible updates to `$s` are the three kinds specified, i.e., there are no updates such as `$s=$t`, then the linear-time query `$s.len()` can be replaced with an efficient retrieval from `$r`, and efficient maintenance can be done at each update as specified.

```
inv py{ $r } = py{ $s.len() }

at py{ $s = set() }
do py{ $r = 0 }

at py{ $s.add($x) }
do before py{
  if $x not in $s:
    $r = $r + 1
}
at py{ $s.remove($x) }
do before py{
  if $x in $s:
    $r = $r - 1
}
```

Figure 1. An invariant rule for set size.

The `inv` clause denotes an invariant between a result variable and a query computation. An `at` clause denotes an update to the values that the query depends on. A `do` clause below an `at` clause (or `inv` clause) denotes maintenance of the query result at the update (or query), which can be done before or after the update (or query). The notation `py{}` indicates that the enclosed text is in Python. The symbol `$` precedes a meta variable that can be instantiated to any program variable or program segment in general.

In general, a rule may also specify, below the query and each update, conditions on the query or update, using an `if` clause, and declarations (with their scopes) needed for the maintenance, using a `de` clause. For convenience, maintenance at an update may also be done in place of the update, using a `do instead` clause. The general form of an invariant rule is given in Figure 2. where *computation*, *result*, *update*, *declaration*, and *maint* are program text, except that they may contain meta variables; and *condition* and *scope* are a Boolean expression and a scope expression, respectively, in the rule language. Scope expressions are of the form **global**, **package** *packagename*, **class** *classname*, or **method** *methodname*.

inv *result* = *computation*
(**if** *condition*+)?
(**de** ((**in** *scope* :)? *declaration*+)+)?
(**do** *maint*? (**before** *maint*)? (**after** *maint*)?)?
(**at** *update*
(**if** *condition*+)?
(**de** ((**in** *scope* :)? *declaration*+)+)?
(**do** *maint*? (**before** *maint*)? (**after** *maint*)?
(**instead** *maint*)?)?)+

Figure 2. General form of an invariant rule.

The semantics of applying an invariant rule is: (1) match a computation in the program against *computation*, match all possible updates to the values on which the computation depends against some *update*, and check all corresponding *condition*s, and (2) if these succeed, replace all occurrences of the computation with the corresponding *result*, add corresponding *declaration*s in the specified scope, and add corresponding *maint* code before or after the computation and before, after, or in place of the updates. For convenience, a declaration of an existing method, class, or module inserts the specified body at the beginning of the existing body of the method, class, or module, respectively.

Applying a rule requires automatic detection of all possible updates to values on which the computation depends. We use powerful static analyses—control flow, data type, and alias analysis [Gorbovitski et al. 2010]—to minimize the set of possible updates and insert runtime checks to confirm them.

To use invariant rules for optimization, the overall algorithm repeatedly applies rules to expensive queries and updates in the given program until no rule applies. The order that rules are applied in follows dependencies among the queries.

The advantage of using an invariant rule to maintain the result of a query, such as the set size, is that all maintenance code is specified declaratively in one rule, and the rule is applied automatically, without the rule's author needing detailed knowledge of the program. This contrasts maintenance code specified in multiple rules that must be coordinated for transformations, or manually inserted at scattered updates throughout the program, possibly refactored to improve the program. For simple queries such as the set size, manual code insertion and refactoring is not too difficult, but it becomes a serious challenge when maintaining an invariant efficiently requires knowing the internals of multiple classes [Gorbovitski et al. 2008; Liu et al. 2005], or when the class itself is complex, such as the `bitTorrent` class of the BitTorrent application.

Instrumentation rules. We slightly extend the rule language above to support instrumentation, using instrumentation rules and pure instrumentation rules. Pure instrumentation rules preserve program semantics. An instrumentation rule is of the same form as an invariant rule, with two exceptions:

1. The clause **inv** *result* = *computation* is replaced with `instrumentation` or `pure instrumentation`, indicating that the rule is not for maintaining an invariant, but for instrumentation or pure instrumentation, respectively.
2. Pure instrumentation rules cannot have `do instead` clauses, meaning that all maintenance code must not replace existing code, but be inserted before or after existing code.

The semantics of applying an instrumentation rule differs from applying an invariant rule in two ways. First, the `do` clause below the `instrumentation` or `pure instrumentation` clause inserts code before or after the entire program, instead of before or after the query in the `inv` clause as for invariant rules. Second,

applying a pure instrumentation rule automatically checks that inserted code does not update existing variables and fields in the program, instead of detecting all possible updates to the values that a query depends on as for invariant rules. This checking uses conservative static analysis first and dynamic checking for the remaining updates. This ensures that pure instrumentation rules do not change program semantics other than the extra time and space for running the inserted code.

An instrumentation rule is applied only once to the given program, and not applied again to the transformed parts—this guarantees termination. This contrasts invariant rules, which may be repeatedly applied to the transformed parts—this stops because invariant rules are designed to reduce program complexities when possible and have clear limits.

Running example. We use instrumentation and incrementalization of BitTorrent, version 4.9.3, as a running example.

BitTorrent (`http://download.bittorrent.com/dl/`) is a peer-to-peer distributed file sharing protocol. When multiple peers download the same file concurrently, they can relay data to each other, making it possible for the file source to support large numbers of downloaders with only a modest increase in its load. Each peer downloads pieces of a file from other peers, and then reassembles the original file from the pieces. The set of peers that a peer communicates with is called its peer horizon.

Each piece is sent as a sequence of packets. Once a piece is completely received, the peer verifies that the piece arrived without errors, by using an SHA1 checksum sent in a bootstrapping file that contains the checksum of each piece of the file being distributed. If the piece contains errors, the peer marks the sender of the piece as untrustworthy, and attempts to retrieve the piece from another peer.

3. Instrumentation of BitTorrent

We instrument the BitTorrent peer to rank peers, giving lower ranks to peers that sent or received mismatched data packets. Doing this efficiently allows us to quickly detect bad peers or peers connected by bad links. In BitTorrent without instrumentation, such detection requires the peer to receive one complete piece from another peer and thus has a delay, because checking is done at the piece level, rather than the packet level.

Figures 3 and 4 together show the complete instrumentation rule. An instrumented BitTorrent peer, in Figure 4, calls method `process` to (1) record history—send a notification packet to all peers in its peer horizon when it receives or sends a data packet, and record the notification packets received, (2) analyze recorded history—compute the ranks of all peers in the peer horizon to reflect matches between the data packets sent and received, and (3) act on the analysis result—sort and write out the list of peers in order of high to low ranks.

Recording history. When the BitTorrent peer receives or sends a data packet `p`, in the middle block of method `process` in Figure 4, it sends a notification packet to all peers in its peer horizon. This is done by calling method `send_notification_packet` that is defined in Figure 3, passing in value `"s"` or `"r"` indicating whether the peer was sending or receiving the data packet, and information about the packet `p`.

When the BitTorrent peer receives a notification packet, in the last block of method `process` in Figure 4, it decodes the packet and stores the decoded information in `$sent` or `$recv` based on value `"s"` or `"r"`. This is done by calling method `receive_notification_packet` that is defined in Figure 3.

Analyzing recorded history. Method `compute_rank` in Figure 4 uses `$sent` and `$recv` to compute the rank of each peer in the peer horizon. For each peer `peer`, uniquely identified by its address ip, it computes `match`, the number of data packets sent by or received by the peer and whose sending and receiving payloads match, i.e.,

$$\texttt{match} = |\{p : p \in \texttt{\$sent} \cap \texttt{\$recv}, p.\texttt{src} = ip \vee p.\texttt{dst} = ip\}|$$

```
pure instrumentation

de in global py{
   import scapy  #socket module from http://www.secdev.org

   #called when a data packet is sent
   def send_notification_packet(peer, type, p):
     ... #send event type and info about packet p to
         #target peer using scapy over UDP on port 555
   $sent = set()  # set of all data packets sent
   $recv = set()  # set of all data packets received
   #called when a notification packet is received
   def receive_notification_packet(bytestring):
     ... #receive a bytestring, decode it, and insert
         #result in $sent or $recv, respectively
}
```

Figure 3. Instrumentation rule clauses for sending and receiving notification packets.

```
de in global py{
   from collections import defaultdict  #standard library
}

de in class bitTorrent py{

   #insert instrumentation at the start of method __init__
   def __init__(self):
     #start sniffing for packets sent/recv'd by current proc.
     #when a packet is sniffed, self.process is called on it
     scapy.sniff(prn = self.process)
     self.rank = defaultdict(float)  #rank for each peer
     self.packet_count = 0  #num. notif. packets received

   def process(self, packet):
     #if packet is UDP or TCP packet, decode packet into p
     if UDP in packet or TCP in packet:
       p = packet[UDP] if UDP in packet else packet[TCP]

       if p.port in self.portrange:  #if p is a data packet
         if p.src==self.ip_addr:     #if sending p
           for peer in self.peers:   #notify peer horizon
             send_notification_packet(peer, "s", p)
         if p.dst==self.ip_addr:     #if receiving p
           for peer in self.peers:   #notify peer horizon
             send_notification_packet(peer, "r", p)

       if p.port==555:               #if p is notif. packet
         if p.dst==self.ip_addr:     #if receiving p
           receive_notification_packet(p.payload) #recording
           self.compute_rank()       #analysis
           self.sort_and_print()     #action
     #otherwise, we sniffed an unknown packet; do nothing

   def compute_rank(self):           #for analysis
     for peer in self.peers:
       match = len(set(p for p in intersect($sent,$recv) if
                 p.src==peer.ip_addr or p.dst==peer.ip_addr))
       total = len(set(p for p in union($sent,$recv) if
                 p.src==peer.ip_addr or p.dst==peer.ip_addr))
     self.rank[peer] = 1.0 if total==0 else 1.0*match/total

   def sort_and_print(self):         #for action
     self.packet_count += 1
     if self.packet_count % 1000 == 0:
       ... #call library functions for sorting and printing
}
```

Figure 4. Rule clauses to instrument BitTorrent peer to process packets received, compute ranks, and print sorted peers.

and it computes $total$, the number of all data packets sent by or received by the peer, i.e.,

$$\texttt{total} = |\{p : p \in \texttt{\$sent} \cup \texttt{\$recv}, p.\texttt{src} = ip \vee p.\texttt{dst} = ip\}|$$

The peer's rank is computed as `match` divided by `total`, i.e.,

$$\texttt{rank} = \texttt{match}/\texttt{total}$$

Higher ranks indicate better peers. A peer's rank is 1 if all data packets it sent and received match, i.e., no packet is sent but not received, received but not sent, or modified in transit.

Acting on analysis results. Method `sort_and_print` in Figure 4 does the sorting and printing for every 1000 notification packets received.

Overhead caused by instrumentation. The overhead caused by instrumentation is shown in the first row of Table 1. It is for (1) sending notification packets to all peers in the peer horizon, whenever a peer sends or receives a data packet, and (2) executing the queries that compute `match` and `total` for all peers in the peer horizon, whenever a peer receives a notification packet. This takes O($(S+R)^2 \times H$) expected time, because a total of O($S+R$) data and notification packets are sent and received by each peer, each packet sent or received has a cost factor of O(H), and computing `match` and `total` takes O($S+R$) expected time using hashing. The space used by the added code is O($S+R$) for storing `$sent` and `$recv`.

4. Decomposition and incrementalization

Complex queries in instrumentations and in programs are not only expensive, but often repeated while the values they depend on change. For example, in the instrumentation for BitTorrent, the queries for computing `match` and `total` take O($S+R$) time, and the query is repeated for each notification packet received.

We optimize these queries by storing the query results and computing the results incrementally as the values the queries depend on change. For example, for BitTorrent instrumentation, we maintain, for each peer in the peer horizon, the values of `match` and `total` incrementally as `$sent` and `$recv` are updated.

We could use a previously studied method [Liu et al. 2005, 2009] to incrementalize expensive queries. It incrementalizes each query in a basic form using an invariant rule; the transformations replace the query with a retrieval of the query result from the result variable, and insert code to maintain the query result at all places that update the values that the query depends on. For nested queries, the effect is that the innermost query in a basic form is incrementalized first; after this the query is replaced by a retrieval of its result from a variable, the outer query that is then in a basic form is incrementalized next; this continues until the outermost query is incrementalized.

This previous method of repeatedly applying invariant rules has three drawbacks: (1) the overall result of incrementally computing a nested query is difficult to understand because it is scattered in many places in the final transformed program, (2) optimizations enabled by incrementalization are hard to perform on the often large and complex transformed program, and (3) repeatedly applying invariant rules is expensive because complex control flow, data type, and alias analyses of the entire program are required before applying each rule.

To overcome these drawbacks, our method automatically composes the transformation rules and optimizes the composed rules before applying the optimized composed rules to the program. To prepare for composition, the method first decomposes nested queries into subqueries in basic forms, which contain no nested subqueries, and uses previous methods [Liu et al. 2006; Rothamel and Liu 2008] to derive invariant rules for incrementally computing the subqueries.

Decomposing nested queries. The parameters of a query are the variables used by the query but defined outside the query.

Decomposing nested queries has three steps. Step 1 extracts subqueries following the innermost, leftmost-first dependency order of computation. That is, if a subquery is contained inside another subquery, then the inner one is extracted first; if neither of two subqueries is contained inside the other, then the left one is extracted first. Step 2 introduces, for each subquery, a map from tuples of values of the subquery parameters to subquery results. Step 3 rewrites the original query to use this map in place of the subquery.

instrumented BitTorrent variant	time	space
use no inv. rules	O($(S+R)^2 \times H$)	$S+R$
use separate inv. rules	O($(S+R) \times H$)	$5(S+R)$
use composed inv. rules	O($(S+R) \times H$)	$5(S+R)$
use opt. composed inv. rules	O($(S+R) \times H$)	$S+R$

S and R are the sizes of `$sent` and `$recv`, respectively.
H is the maximal number of peers in the horizon of any given peer.

Table 1. Time and space overhead caused by instrumentation.

For example, for the query for computing `match`:

```
len(set(p for p in intersect($sent,$recv)
        if p.src==peer.ip_addr or p.dst==peer.ip_addr))
```

Step 1 first extracts the inner-most subquery `intersect($sent, $recv)`, Step 2 introduces a new map `$I` to store the result of this subquery, and Step 3 replaces the subquery in the original query with `I[($sent,$recv)]`, yielding

```
$I[($sent,$recv)] = intersect($sent,$recv)
match             = len(set(p for p in $I[($sent,$recv)]
                            if (p.src==$ip or p.dst==$ip))
```

Repeating this procedure until we reach the outermost query, we obtain new maps `$I`, `$P`, and `$M` that store the intersection, the selected set for each peer, and the results of the query for `match`, respectively; and we replace the original query by `$M[($sent,$recv,peer.ip_addr)]`.

```
$I[($sent,$recv)]     = intersect($sent,$recv)
$P[($sent,$recv,$ip)] = set(p for p in $I[($sent,$recv)]
                            if (p.src==$ip or p.dst==$ip))
$M[($sent,$recv,$ip)] = len($P[($sent,$recv,$ip)])
```

Parameters that throughout the lifetime of the program are bound to a single object are unnecessary and thus removed. For the example above, parameters `$sent` and `$recv` are removed, yielding the subqueries in Figure 5, and the original query for computing `match` is then replaced by `$M[peer.ip_addr]`. The

```
$I      = intersect($sent,$recv)
$P[$ip] = set(p for p in $I if p.src==$ip or p.dst==$ip)
$M[$ip] = len($P[$ip])
```

Figure 5. Result of decomposing `match` query.

query for computing `total` can also be decomposed into three subqueries, one each for the union, the selection, and the result.

Deriving invariant rules for subqueries. For each subquery, we use previous methods [Liu et al. 2006; Rothamel and Liu 2008] to derive invariant rules for incrementally maintaining the query result under each kind of update to a parameter of the query. The methods work for large classes of queries and updates. For example, for incrementally maintaining `$I`, `$P`, and `$M`, the resulting invariant rules in Figure 6 are derived. Note that the parameter `$ip` in a result expression allows the query result to be looked up in the result map for any `$ip` given. Similar rules for incrementally computing subqueries for `total` can also be derived.

Overhead caused by instrumentation using separate invariant rules. The overhead caused by instrumentation after incrementalization using separate invariant rules is as shown in the second row of Table 1. The time complexity is O($(S+R) \times H$), because each piece of maintenance code inserted takes constant time, and retrieving query results from all three maps also takes constant time, and thus the overhead is constant for each peer in the peer horizon for each packet sent or received. The space complexity is bounded by $(S+R) \times 5$ because, besides storing `$sent` and `$recv`, we also store `$I` and `$P` for computing the query for `match` and two similar variables for computing the query for `total`, and the space for each of these four maps is bounded by $S+R$; the result map `$M` for the query for `match` and the result map for the query for `total` take significantly less space and thus are omitted.

```
inv py{ $I } = py{
    intersect($sent,$recv)
}
de in class bitTorrent py{
   def __init__(self):
     $I = set()
}
at py{ $sent.add($p) }
do before py{
   if $p in $recv:
     if $p not in $I:
       $I.add($p)
}
at py{ $recv.add($p) }
do before py{
   if $p in $sent:
     if $p not in $I:
       $I.add($p)
}
```

```
inv py{ $P[$ip] } = py{
    set(p for p in $I if
          p.src==$ip or p.dst==$ip)
}
de in class bitTorrent py{
   def __init__(self):
     $P = defaultdict(set)
}
at py{ $I.add($p) }
do before py{
   if $p not in $P[$p.src]:
     $P[$p.src].add($p)
   if $p not in $P[$p.dst]:
     $P[$p.dst].add($p)
}
at py{ $I.remove($p) }
do before py{
   if $p not in $P[$p.src]:
     $P[$p.src].remove($p)
   if $p not in $P[$p.dst]:
     $P[$p.dst].remove($p)
}
```

```
inv py{ $M[$ip] } = py{
    len($P[$ip])
}
de in class bitTorrent py{
   def __init__(self):
     $M = defaultdict(int)
}
at py{ $P[$ip].add($p) }
do before py{
   $M[$ip] += 1
}
at py{ $P[$ip].remove($p) }
do before py{
   $M[$ip] -= 1
}
```

Figure 6. Invariant rules for maintaining the results of subqueries for computing `match` in `$I`, `$P`, and `$M`. Clauses for handling removals from `$sent` and `$recv` are symmetric to clauses for handling addition and are omitted for brevity.

5. Composition and optimization

We describe composition of invariant rules and optimization of composed rules. We then discuss composition of instrumentation rules with invariant rules. Our composition and optimization preserve program semantics, i.e., applying a composed rule or an optimized composed rule yields programs that have the same semantics as programs obtained by applying individual rules; however, applying an optimized composed rule may yield more efficient programs due to optimizations performed on the composed rules.

5.1 Composition of rules

Given a nested query q decomposed into a sequence of subqueries, they form a sequence of invariants $r_1 = q_1, r_2 = q_2, \ldots, r_n = q_n$, where the value of original query q equals the result r_n of the last query with its parameters instantiated to the corresponding parameters in q. For example, the subqueries for computing `match`, in Figure 5, form three invariants, and the value of the original query equals the result `$M[$ip]` of the third subquery with its parameter `$ip` instantiated to `peer.ip_addr`.

For each invariant $r_i = q_i$ in the sequence, there is a corresponding invariant rule R_i of the form `inv` $r_i = q_i$ B_i, where B_i is the body of the rule. B_i may have multiple `at`, `if`, `de`, and `do` clauses. Composition must combine code patterns in all `inv`, `at`, `de`, and `do` clauses, as described below. The conditions in `if` clauses can be evaluated or simplified using static analysis during the composition. Composition produces a single rule whose invariant is $r_n = q'_n$ such that q'_n, with its parameters instantiated to the corresponding parameters in the original query q, is syntactically identical to q.

The composition algorithm builds a composed rule up starting from the first rule—the rule for the innermost subquery of the original query. The construction produces a sequence of rules $R'_1, R'_2, \ldots, R'_n$, where R'_i is the result of composing rules R_1 to R_i. As the base case, R'_1 is identical to R_1. At the end, R'_n is the desired rule for q. We give the precise algorithm below for the case that the sequence of subqueries are strictly nested; it is straightforward to extend it to handle multiple independent subqueries inside an enclosing query. We use $t_1[v \mapsto t_2]$ to denote t_1 with each occurrence of v replaced with t_2.

$$
\begin{array}{ll}
q'_1 = q_1;\ \ B'_1 = B_1;\ \ R'_1 = R_1 & (1)\\
\texttt{for}\ i = 1\ \texttt{to}\ n-1 & (2)\\
\quad q'_{i+1} = q_{i+1}[r_i \mapsto q'_i] & (3)\\
\quad B'_{i+1} = \texttt{transform}(B'_i, R_{i+1}) & (4)\\
\quad R'_{i+1} = \textbf{inv}\ \ r_{i+1} = q'_{i+1}\ B'_{i+1} & (5)
\end{array}
$$

Applying the substitution $[r_i \mapsto q'_i]$ to q_{i+1} in line (3) reconstructs part of the structure of the original nested query, because this substitution reverses the replacement of q_i with r_i when extracting q_i from q_{i+1} during the decomposition. This substitution is valid because R'_i ensures its invariant $r_i = q'_i$. To ensure that R'_{i+1} also maintains this invariant, the body B'_i of R'_i is used in line (4) as the basis for the body B'_{i+1} of R'_{i+1}.

To ensure that R'_{i+1} also maintains the invariant of R_{i+1}, the transformation specified by B_{i+1} is applied to the maintenance code in B'_i. Specifically, `transform`(B'_i, R_{i+1}) in line (4) returns the result of that application. Following the semantics for applying invariant rules, `transform` first checks whether every update to parameters of q_{i+1} in B'_i matches some *update* pattern in B_{i+1}. If so, declarations and maintenance code in B_{i+1} are inserted in B'_i as specified by the `de` and `do` clauses in B_{i+1}. If not, `transform` aborts, which causes the composition algorithm to abort.

Note that the transformation defined by B_{i+1} is applied only to code in B'_i. If we did not use rule composition, it would be applied to the entire subject program. To ensure that applying it only to B'_i gives the same result as applying it to the entire subject program, `transform`(B'_i, R_{i+1}) checks that every update pattern in B_{i+1} updates only the query parameter that is the query result r_i introduced by R'_i and hence would not match any other update in the subject program. If this condition is not satisfied, `transform` aborts.

When applying R_{i+1} to the maintenance code in B'_i, `transform`(B'_i, R_{i+1}) needs alias information to identify possible updates to the query parameters in R_{i+1}. The two checks above imply that the only such query parameter is the result variable of R_i; updates to index variables in R_i, if any, do not matter because the query result can be looked up for any index values. Standard alias analysis cannot be used here, because it requires the whole program. Instead, `transform`(B'_i, R_{i+1}) checks whether B'_i contains assignments that could create aliases to that result variable. If so, the call to `transform` aborts; otherwise, it proceeds knowing that result variable has no aliases.

To summarize, the algorithm succeeds for rules that obey the following: (1) every update in the maintenance code of the rule for an inner query is handled by an update pattern of the rule for the enclosing query, (2) every update in the rule for an outer query updates only query parameters that are the query results in the rules for the enclosed queries, and (3) the maintenance code in a rule does not create aliases to the result variable. The algorithm is correct because each iteration of its for-loop ensures that an invariant is preserved. Figure 7 shows the result of composing the three rules in Figure 6.

```
at py{ $sent.add($p) }
do before py{
   if $p in $recv:
     if $p not in $I:
       if $p not in $P[$p.src]:
         $M[$p.src] += 1
         $P[$p.src].add($p)
       if $p not in $P[$p.dst]:
         $M[$p.dst] += 1
         $P[$p.dst].add($p)
       $I.add($p)
}
```

```
at py{ $recv.add($p) }
do before py{
   if $p in $sent:
     if $p not in $I:
       if $p not in $P[$p.src]:
         $M[$p.src] += 1
         $P[$p.src].add($p)
       if $p not in $P[$p.dst]:
         $M[$p.dst] += 1
         $P[$p.dst].add($p)
       $I.add($p)
}
```

```
at py{ $sent.remove($p) }
do before py{
   if $p in $recv:
     if $p in $I:
       if $p in $P[$p.src]:
         $M[$p.src] -= 1
         $P[$p.src].remove($p)
       if $p in $P[$p.dst]:
         $M[$p.dst] -= 1
         $P[$p.dst].remove($p)
       $I.remove($p)
}
```

```
at py{ $recv.remove($p) }
do before py{
   if $p in $sent:
     if $p in $I:
       if $p in $P[$p.src]:
         $M[$p.src] -= 1
         $P[$p.src].remove($p)
       if $p in $P[$p.dst]:
         $M[$p.dst] -= 1
         $P[$p.dst].remove($p)
       $I.remove($p)
}
```

Figure 7. Result of composing the rules in Figure 6 for computing `match`. The `inv` and `de` clauses are not shown; they are the same as in the optimized rule on the left of Figure 8, except that, in the `de` clause, the definition of `__init__` also contains `$P = defaultdict(set)` and `$I = set()`.

5.2 Optimization of composed rules

Optimizing the maintenance code in invariant rules, before applying the rules, conveniently allows the invariants maintained by the rules to be exploited for optimization. While these invariants could be made available to an optimizer running on the transformed program, it is much more difficult and less efficient to optimize the transformed program, which is typically much larger, than the rules.

Optimizing the maintenance code is especially useful for rules constructed by composition, because composition of separate rules may introduce redundant or dead computations. Our method repeatedly eliminates redundant computations and dead computations in the composed rules until no more can be eliminated.

Eliminating redundant computations. Composing rules derived for separate subqueries may produce, in the composed maintenance code, redundant computations, i.e., computations that are unnecessary for producing the desired result. For queries over sets, the redundant computations are dominantly redundant membership tests, i.e., membership tests that can be statically simplified to be `true` or `false`. We first show an example before describing membership test simplification in general.

Consider the code segment on lines 4-6 in the third column of Figure 7:

```
if $p in $I:
  if $p in $P[$p.dst]:
    $M[$p.dst] -= 1
```

First, using the invariant about `$P` from Figure 5:

```
$P[$ip] = set(p for p in $I if p.src==$ip or p.dst==$ip)
```

the membership test `$p in $P[$p.dst]` is replaced with its equivalent, yielding the following rewritten code segment:

```
if $p in $I:
  if $p in $I and ($p.src==$p.dst or $p.dst==$p.dst):
    $M[$p.dst] -= 1
```

Then, the resulting conjunction is simplified; the first conjunct becomes `true` because it equals the condition of the encosing `if` statement and is in the `true` branch of the statement:

```
if $p in $I:
  if true and ($p.src==$p.dst or $p.dst==$p.dst):
    $M[$p.dst] -= 1
```

then the second equality is symbolically evaluated to `true`:

```
if $p in $I:
  if true and ($p.src==$p.dst or true):
    $M[$p.dst] -= 1
```

and further symbolic evaluation of Boolean expressions yields:

```
if $p in $I:
  if true:
    $M[$p.dst] -= 1
```

In general, our method simplifies membership tests of the form v `in` r such that $r = q$ is an invariant generated during query decomposition and q has the form `set(`x `for` x `in` S `if` c`)`. This is done in two steps.

Step 1 replaces a membership test v `in` r with the equivalent v `in` S `and` $c[x \mapsto v]$, where $c[x \mapsto v]$ denotes c with all occurrences of x replaced with v.

Step 2 simplifies the conjuncts from Step 1 by repeatedly applying (a) simplification in context, and (b) symbolic evaluation of primitives, until no more simplification can be done.

For (a), if any conjunct simplifies to an expression that is the condition of an enclosing `if` statement or the negation of the condition, and if variables used by the conjunct are not updated between the condition and the conjunct, then the conjunct is replaced with `true` or `false`, respectively. For (b), standard symbolic evaluation is used, e.g., for any expression e, e `and true` simplifies to e; and for expression e without side effect, e`==`e simplifies to `true`. Checking updates and side effects uses alias analysis conservatively as in composing rules. If (a) or (b) replaces any conjunct with a Boolean constant, then the original membership test is simplified; otherwise, the membership test is left unchanged.

It would be difficult to perform this optimization based purely on analysis of the transformed program because Step 1 would require re-discovering the invariant of the invariant rule.

Eliminating dead computations. Dead computations include dead branches, i.e., branches that will never be executed; dead variables, i.e., variables that will never be used; and updates to dead variables.

If the condition in an `if` statement is a Boolean constant, usually as a result of membership test simplification, then the alternative branch is dead, and the `if` statement is replaced with the reachable branch. For the example above, this optimization replaces

```
if true:
  $M[$p.dst] -= 1
```

with

```
$M[$p.dst] -= 1
```

If the value of a variable that is introduced by an invariant rule is not used in the rule's result (on the left side of the `inv` clause) or in the rule's maintenance code (in `do` clauses), and there are no aliases of the variable, then the variable and all updates to it are dead and thus eliminated. For example, after repeatedly applying membership test simplification to the rules in Figure 7, variables `$I` and `$P` are dead, so these variables and updates to them are eliminated.

Applying these optimizations to the composed rule in Figure 7 for maintaining `match` and the similar composed rule for maintaining `total`, we obtain the optimized composed rules in Figure 8. Note how much easier the `at` clauses in these rules are to understand than those in Figure 7.

Overhead caused by instrumentation using optimized composed rules. The optimized composed invariant rule for computing `match` does not use `$I` and `$P`. Similarly, the optimized composed invariant rule for computing `total` does not introduce maps maintaining the union and peer selection. Thus, the optimizations eliminate four maps, each of size O($S + R$). This is reflected in the improved space complexity in the last row in Table 1.

```
inv py{ $M[$ip] } = py{
    len(set(p for p in intersect($sent,$recv)
              if p.src==$ip or p.dst==$ip))
}
de in class bitTorrent py{
   def __init__(self):
     $M = defaultdict(int)
}
at py{ $sent.add($p) }
do before py{
   if $p in $recv:
     if not ($p in $sent):
       $M[$p.src] += 1
       $M[$p.dst] += 1
}
at py{ $recv.add($p) }
do before py{
   if $p in $sent:
     if not ($p in $recv):
       $M[$p.src] += 1
       $M[$p.dst] += 1
}
```

```
inv py{ $T[$ip] } = py{
    len(set(p for p in union($sent,$recv)
              if p.src==$ip or p.dst==$ip))
}
de in class bitTorrent py{
   def __init__(self):
     $T = defaultdict(int)
}
at py{ $sent.add($p) }
do before py{
   if $p not in $recv:
     if not ($p in $sent):
       $T[$p.src] += 1
       $T[$p.dst] += 1
}
at py{ $recv.add($p) }
do before py{
   if $p not in $sent:
     if not ($p in $recv):
       $T[$p.src] += 1
       $T[$p.dst] += 1
}
```

Figure 8. Optimized composed rules for maintaining `match` and `total`.

5.3 Composing instrumentation rules with invariant rules

Our system composes instrumentation rules with invariant rules by applying invariant rules, including composed and optimized composed invariant rules, to the code in instrumentation rules, before applying the instrumentation rules to a subject program. This allows expensive queries in instrumentation code to be incrementalized before the instrumentation code is inserted in a subject program. When applying an invariant rule to the code in an instrumentation rule, the analysis and transformations are done in the same way as when applying an invariant rule to the maintenance code of another invariant rule when composing invariant rules.

This composition is not essential, but it reduces the overall transformation time. Applying an invariant rule to an instrumentation rule, and then applying the resulting rule to the subject program, requires one analysis of the code in the instrumentation rule and one analysis of the subject program. Sequentially applying the instrumentation rule, and then the invariant rule to the subject program, requires two analyses of the subject program. The former increases performance, because the code in an instrumentation rule is typically much smaller than the subject program, and because the alias analysis used to analyze code in instrumentation rules is less sophisticated, and hence cheaper, than the alias analysis used to analyze subject programs.

6. Experiments

We have implemented the composition and optimization method by extending InvTS [Gorbovitski et al. 2010; Liu et al. 2005, 2009], a system for applying invariant rules, performing powerful analysis, and deriving classes of invariant rules. Our implementation handles the entire Python 2.5 language. The generated, optimized composed rules make the effect of transformations much easier to see, as discussed. We then performed experiments to confirm that our method also increases the efficiency of the transformed program and reduces the transformation time, as described below.

We used three diverse applications: BitTorrent, a NetFlow query tool, and Constrained RBAC. For experiments, we automatically transformed each application using three transformation variants:

1. Application of separate rules, in dependency order.
2. Composition of rules, followed by application of the composed rule.
3. Composition and optimization of rules, followed by application of the optimized rule.

For each variant, we measured the size of the application before and after the transformation, the times it took to compose the rules and to optimize the composed rules, the time it took InvTS to apply the rules, and other quantities about the transformed and original programs. All programs were written in Python and all experiments were run under Python 2.6.1. Table 2 summarizes the results, explained below.

6.1 BitTorrent

We instrumented BitTorrent and optimized the instrumentation as described in the running example. When the five rules totaling 171 lines are separately applied to the BitTorrent peer, the code size increases from 41,162 to 41,374 lines, a difference of 212 lines.

Composition and optimization of rules. To evaluate the efficiency of the BitTorrent peers instrumented using each of the three transformation variants, we performed experiment that measured the number of notifications stored by the instrumentation, the number of set operations performed by the instrumentation, the CPU usage, and the total network usage. During each experiment, we transferred a 1GB file from a BitTorrent peer to 29 other BitTorrent peers over a 100 MBit link. Each peer was on a virtual machine running Ubuntu 9.04 with 1GB of RAM and a single core of a Xeon L5430 @ 2.66GHz provisioned to it. Because the peers were never CPU-bound, CPU under-provisioning was not an issue. Table 2(a) summarizes the results.

Instrumented using composed but not optimized rules, the BitTorrent peers stored 93 million notifications, and performed 190 million additional set operations. Using optimized composed rules eliminated intermediate query results and thereby about two thirds of the storage overhead, reducing the number of stored notifications to 25 million, and the number of additional set operations to 59 million.

The BitTorrent peers instrumented using separate rules and using composed rules both have CPU usage that is 7% higher than the CPU usage of the original BitTorrent peers, due to the maintenance of intermediate query results by both of them. In contrast, using optimized composed rules eliminated these intermediate results and reduced the CPU usage to be within 0.5% of the original BitTorrent peers.

Because the experiments were ran on top-of-the-line machines connected by only 100 MBit links, none of the BitTorrent variants were CPU bound, and thus the CPU overhead did not affect the total time to transfer the file to 29 peers, which was about 220 seconds. Since the CPU utilization was about 50% even with 100 MBit links, we estimate that if one was to upgrade the links to the currently industry-standard 1 GBit, the peers would become CPU bound, and thus the total time to transfer the file would be noticeably higher for the BitTorrent peers instrumented using separate rules and composed rules than for either the original BitTorrent peers or the peers instrumented using optimized composed rules.

(a) BitTorrent

	# LOC before	# LOC after	#rules	composition time (s)	optimization time (s)	**rule application time (s)**	**notifications stored (millions)**	**extra set ops. (millions)**	**CPU usage**	**total network**
Original	41,162	41,162	-	-	-	-	-	-	48.6%	32.1GB
Separate rules	41,162	41,374	6	-	-	2998	96.3	193.3	56.1%	33.1GB
Composed rules	41,162	41,374	6	2.9	-	2320	93.1	189.6	56.9%	32.7GB
Opt. composed	41,162	41,331	6	2.8	3.5	2261	25.0	58.8	49.1%	33.3GB

(b) NetFlow query tool

	# LOC before	# LOC after	#rules	composition time (s)	optimization time (s)	**rule application time (s)**	**total processing time (s)**	**throughput (packets/s)**	
Original query	64	64	-	-	-	-	>600	81	
Separate rules	64	105	5	-	-	21.1	33.1	302,114	
Composed rules	64	105	5	1.0	-	15.3	32.8	304,878	
Opt. composed	64	75	5	1.0	0.4	15.4	19.9	502,512	

(c) Constrained RBAC

	# LOC before	# LOC after	#rules	composition time (s)	optimization time (s)	**rule application time (s)**	**# `inv` clauses applied**	
Separate rules	381	2,183	21	-	-	257.4	38	
Composed rules	381	2,183	21	1.1	-	44.2	27	
Opt. composed	381	2,183	21	1.1	0.5	44.8	27	

Table 2. Summary of rule composition and optimization experiments.

Rule application time. Table 2(a) shows that applying separate rules takes the longest time: 2,998 seconds. Applying composed rules takes 2,320 seconds, after taking less than 3 seconds to compose the rules, a net savings of 675 seconds. Optimizing the composed rule takes under 4 seconds, and reduces rule application time to 2,261 seconds, a further gain of 55 seconds.

Effects of instrumentation on free-riding clients. There are non-specification-adhering modifications to BitTorrent clients that attempt to get around the BitTorrent choking feature that prevents specification-adhering clients from sending data to free-riding clients [Moor 2006]. One such modification has the peer start sending out pieces of the torrent before the peer has fully downloaded them. This self-promotion causes no harm when there are few or no network errors, but it makes the swarm susceptible to swarm poisoning—wide propagation of pieces corrupted by network errors—when network errors increase.

To measure the effect of swarm poisoning, we transferred a 1GB file from a BitTorrent peer to 29 other BitTorrent peers over a 100 MBit link, with 3 of 29 peers having a 10% error rate. This took 438 seconds and a total bandwidth of 93.1GB. This is over 2 times as long, and a factor of 3 increase in total bandwidth used, compared to the specification-adhering BitTorrent swarm. Note that this is 10% error rate in 10% of the peers, so only 1% overall error rate.

To combat swarm poisoning, we modify our BitTorrent instrumentation rule to use the computed ranks to let the peer avoid connecting to peers with low ranks. The modified rule changes the BitTorrent metric for selecting peers, stored field `goodness` of each peer, to prefer peers with better ranks. The modified rule changes program semantics, so we need to change `pure instrumentation` to `instrumentation` in the rule. We measure the effect of this instrumentation by performing the same experiment as above. The experiment shows that the swarm took 227 seconds and a total bandwidth of 34.2GB to transfer the same 1GB file over a 100 MBit link, which is comparable to the performance of a specification-adhering swarm.

6.2 NetFlow

NetFlow is an IETF-standardized [Claise 2004] network protocol used for analyzing network traffic. In NetFlow, source hosts collect information about their network activity, including information about packets received and sent. They then transmit this information using the NetFlow protocol to a target host, called a NetFlow collector. The collector may analyze the received information on-the-fly, store it for further analysis, or discard it if it cannot cope with the volume of the incoming information.

```
HOSTS = set()
RECV = set()
SENT = set()
for p in generate_netflow_packets():
    if p.is_received:
        RECV.add(p)
    else:
        SENT.add(p)
    HOSTS.add(p.dst)
    query()
```

Figure 9. Pseudocode for the NetFlow query tool.

```
def query():
  for host in HOSTS:
    match = len(set(p for p in intersect(SENT,RECV)
                    if p.dst==host))
    total = len(set(p for p in union(SENT,RECV)
                    if p.dst==host))
    quality[host] = 1.0*match/total
```

Figure 10. The NetFlow query function.

We created a NetFlow query tool based on the collector from the `flowtools` package [Romig 2000]. Figure 9 shows the pseudocode for this tool, where `query` can be any user-specified query function. The tool allows the execution `query` over the sets `SENT`, `RECV`, and `HOSTS`—the set of packets sent by the hosts, the set of packets received by the hosts, and the set of hosts, respectively. The query is executed every time a packet is received or sent.

Queries can be written easily and implemented efficiently using our NetFlow query tool. Figure 10 shows, for ease of explanation, an example query similar to the query for BitTorrent instrumentation. It computes, for each host, the quality of its network connection, defined as the fraction of packets sent to and or received by the host that arrived unchanged, i.e., `match/total`, where `match` is the number of packets that were sent to the host, received by the host, and not modified in transit, and `total` is the total number of packets sent to the host, including packets that were lost or changed.

It is clear that for reasonable performance, the results of `total` and `match` must be incrementally maintained. We do so using our composition and optimization method, by deriving and using five invariant rules. These rules are similar to the rules in Figure 6 for incrementalizing the instrumentation of BitTorrent.

Incrementalization and rule composition. To show the effect of optimizing NetFlow queries using each of the three transformation variants, we ran the original query program and optimized query programs on a set of 10 million packets recorded over the course of about 20 seconds from a saturated Gigabit network with 5 hosts on it. We measure the time to process 10 million packets, and the number of packets processed per second, and we set the time limit for the query program to 600 seconds. The query was run on an Intel i7 920@3.1GHz with 12GB of RAM, running Ubuntu 9.04. Table 2(b) shows the measured results.

The first row shows that running the original query exceeds the time limit of 600 seconds while processing an average of only 81 packets per second. This is because computing `total` and `match` iterates over the entire `SENT` and `RECV` sets every time the NetFlow query is called.

The query program transformed using separate rules or composed rules took approximately 33 seconds to process 10 million packets. In contrast, the query program transformed using optimized composed rules took 19.9 seconds to process the same data. Because the packets were recorded over the course of 20 seconds and there is non-negligible overhead in reading the packets from disk, one can infer that the query program transformed by optimized composed application is capable of running the query in real-time without the need to store the packets to disk. This shows that using optimized composed rules provides very tangible benefits over using separate rules.

Rule application time. Table 2(b) shows that applying separate rules takes the longest time: 21 seconds. Applying composed rules and optimized composed rules takes 15 seconds each, with composition taking an additional 1 second, and optimization taking another 0.5 seconds.

6.3 Constrained RBAC

RBAC is an ANSI-standardized [American National Standards Institute, Inc. 2004] framework for controlling user access to resources based on roles. It can significantly reduce the cost of security policy administration and is increasingly used in large organizations. Core RBAC controls access based on relations among permissions, users, sessions, and roles. Constrained RBAC adds two kinds of constraints:

1. Static Separation of Duty (SSD) constraints. A SSD constraint specifies that a user can be assigned to at most `c` roles from a certain set `R` of roles.
2. Dynamic Separation of Duty (DSD) constraints. A DSD constraint specifies that a session can have at most `c` roles from a set `R` of roles active at the same time.

Mirroring the formal specification of RBAC, we extended the 125-line straightforward implementation of Core RBAC [Liu et al. 2006] into a 381 line straightforward implementation of Constrained RBAC. The queries in Constrained RBAC are much more complex than those in Core RBAC, even after simplification [Liu and Stoller 2007]. For example, the SSD constraints hold if the following universally quantified query returns true.

```
forall u in USERS, [name,c] in SsdNC |
     #{r: r in AssignedRoles(u) | [name,r] in SsdNR } <= c
```

Clearly, a straightforward implementation is extremely inefficient when evaluating expensive queries, including `CheckAccess`, the main query of RBAC.

To improve efficiency, we derived and used 21 invariant rules to optimize the straightforward implementation, incrementalizing all queries in it. Out of the 21 rules, only 7 are unique to Constrained RBAC; the other 14 are the same as the rules used to incrementalize Core RBAC [Liu et al. 2006], and this reuse shows the significant advantage of capturing complex optimizations in rules. For Constrained RBAC, no dead code is eliminated by optimization of composed rules, so the optimized composed rules are identical to the composed rules.

When the straightforward Constrained RBAC program is incrementalized, using all three transformation variants, it becomes 2183 lines of code, a more than 5-fold increase in size. In contrast, when Core RBAC was incrementalized [Liu et al. 2006], it tripled in size to slightly over 400 lines. Incrementalization of queries improves performance asymptotically: for example, `CheckAccess` is improved from $O(roles)$ to $O(1)$, which in our experiments with 100 roles manifests itself as an almost 50-fold speedup.

Rule application time. All experiments were performed on the same machine as in the experiments for the NetFlow query tool. Table 2(c) shows that applying composed rules, compared to applying separate rules, reduces the transformation time by up to a factor of five, from 257 seconds down to 44 seconds . The reason is evident from the "# `inv` clauses applied" column, which shows that when applying composed rules, fewer invariant rules are applied than when applying separate rules. After applying a rule that changes the program, the changed program must be reanalyzed. Thus, applying more separate rules is slower than fewer composed rules, even when both produce the same transformed program.

Correctness. We also experimentally checked that the incrementalization preserved the program semantics, using the same intensive test approach as for Core RBAC [Liu et al. 2006]. Our testing suite randomly generates a sequence of 50 million RBAC operations. It then verifies that the straightforward and incrementalized implementations produce the same results for these operations.

7. Related work

A large amount of work has been done on program transformations and in related areas.

The rule language we use is a slight extension of the invariant rules in [Liu et al. 2009], to support instrumentation rules that were not supported before. It allows concise and convenient specification of program transformations for inserting instrumentation and maintaining invariants. To this end, it supports automatic detection of all program segments that may affect an invariant, and coordinated transformations for all those segments. Pure instrumentation rules ensure that inserted code does not change program semantics. Other powerful program manipulation systems, such as StrategoXL [Visser 2004] and TXL [Cordy 2004], do not provide such support.

Previous work related to invariant rules [Liu et al. 2005, 2009] studied only transformations using repeated application of individual rules, not composition and optimization of rules as in this paper. Our method in this paper makes the result of composition much simpler and easier to understand than before. It also significantly reduces the transformation time, as well as the running time and memory overhead of the instrumentation. Our elimination of dead computations is more powerful than standard compiler optimizations [Aho et al. 2006], because it exploits the results of simplification and symbolic evaluation, as also exploited in partial evaluation [Jones et al. 1993]. In particular, our transformations are based on the semantics of set operations, especially set comprehensions, that have not been studied in partial evaluation before, to the best of our knowledge.

Aspect-oriented programming (AOP) [Kiczales et al. 1997, 2001] also allows code for cross-cutting concerns, such as debugging, to be expressed separately and inserted automatically at a set of matched program points. Connections between AOP and invariants are studied specially [Smith 2007, 2008]. Our work can be viewed as extensions to existing AOP approaches: our rule language has an explicit definition for preserving invariants, to facilitate formal verification, and it provides powerful static analysis, especially for automatically detecting updates, to apply coordinated transformations. Also, existing AOP methods do not help the programmer write code to efficiently maintain the query results—he must figure that out on his own. Finally, existing AOP systems for Python provide a very limited set of join points: `Aspyct.aop` [Antoine 2010] provides just `atCall`, `atRaise`, and `atReturn`, whereas our method provides also the equivalent of pointcuts at field accesses.

Program optimization by incrementalization has been studied for many languages. For example, Acar et al. [Acar 2009] study a combination of change propagation and memoization for ML and

C, which works quite well for recursive algorithms. However, the method requires the programmer to write the program to be transformed using special constructs (e.g., mutable references), and relies on runtime support (e.g., dynamic dependence tracking), with runtime overhead of up to a factor of 18.8 for C [Hammer et al. 2009] and 31.1 for ML [Ruy et al. 2008]. We derive invariant rules by combining a method for sets but not objects [Liu et al. 2006] that is static and a method for sets and objects that is dynamic [Rothamel and Liu 2008]. None of these previous works provides a platform for general and efficient instrumentation, nor do they study composition of transformation rules.

For composing program transformation specifications, there are two approaches. The extensional approach simply concatenates the specifications; applying the resulting specification to a program involves applying the original transformations, one at a time, in the specified order. The intensional approach composes the specifications into a single transformation specification that can be applied in one shot. The extensional approach is used in StrategoXL and TXL. The intensional approach is used in J& [Nystrom et al. 2006], but is limited to specifications that do not depend on static analysis results. Our previous work [Liu et al. 2005, 2009] also uses an extensional approach, but, unlike StrategoXL and TXL, automatically determines the order for applying transformation rules. This paper presents a method for intensional composition without the limitations of J&: we allow the rules to depend on static analysis results, and we also optimize composed rules.

Using our method in complex applications shows the promise of the method. For example, our method makes the instrumentation of BitTorrent significantly easier than manually inserting bookkeeping code. Design and implementation of NetFlow collectors / analyzers that operate at line speeds on Gigabit links (100,000+ packets/sec) is challenging, due to the classic tension between clarity and efficiency, i.e., the desire to let the network administrator write analysis scripts in a declarative manner vs. the desire to have these scripts process hundreds of thousands of packets per second. Some systems allow a degree of customization of the queries that they efficiently execute [Deri 2003; SolarWinds 2009]. Our method can allow such systems to execute even more general queries efficiently. Various implementations of Constrained RBAC exist, such as [Finin et al. 2008; Strembeck 2004; Ventuneac et al. 2003]. We are aware of only one incrementalized implementation—Strembeck's, and it was incrementalized manually. Our method generates efficient implementations automatically.

References

U.A. Acar. Self-adjusting computation: an overview. In *Proc. of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 1–6, 2009.

A. V. Aho, M. S. Lam, , R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2006.

American National Standards Institute, Inc. Role-Based Access Control. ANSI INCITS 359-2004, 2004. Approved Feb. 3, 2004.

d'O. B. Antoine. Aspyct.aop - Python AOP engine. `http://old.aspyct.org/doku.php?id=aspyct`, 2010.

B. Claise. Cisco Systems NetFlow services export version 9. RFC 3954, Internet Engineering Task Force, 2004.

J.R. Cordy. TXL-a language for programming language tools and applications. *Electronic Notes in Theoretical Computer Science*, 110:3–31, 2004.

L. Deri. Passively monitoring networks at Gigabit speeds using commodity hardware and open source software. In *Proc. of the Passive and Active Measurement Conf.*, pages 13–21, 2003.

T. Finin, A. Joshi, L. Kagal, J. Niu, R. Sandhu, W. Winsborough, and B. Thuraisingham. R OWL BAC: representing role-based access control in OWL. In *Proc. of the 13th ACM Symp. on Access Control Models and Technologies*, pages 73–82, 2008.

M. Gorbovitski, T. Rothamel, Y. A. Liu, and S. D. Stoller. Efficient runtime invariant checking: A framework and case study. In *Proc. of the 6th Intl. Workshop on Dynamic Analysis*, pages 43–49, 2008.

M. Gorbovitski, Y. A. Liu, S. D. Stoller, K. T. Tekle, and T. Rothamel. Alias analysis for optimization of dynamic languages. In *Proc. of the 2010 Dynamic Languages Symp.*, pages 12–20, 2010.

M.A. Hammer, U.A. Acar, and Y. Chen. CEAL: a C-based language for self-adjusting computation. In *Proc. of the 2009 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 25–37, 2009. ISBN 978-1-60558-392-1.

N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. of the 11th Europeen Conf. on Object-Oriented Programming*, pages 220–242, 1997.

G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proc. of the 15th European Conf. on Object-Oriented Programming*, pages 327–353, 2001.

Y. A. Liu and S. D. Stoller. Role-based access control: A corrected and simplified specification. In *Department of Defense Sponsored Information Security Research: New Methods for Protecting Against Cyber Threats*. Wiley, 2007.

Y. A. Liu, S. D. Stoller, M. Gorbovitski, T. Rothamel, and Y.E. Liu. Incrementalization across object abstraction. In *Proc. of the 20th Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications*, pages 473–486, 2005.

Y. A. Liu, C. Wang, M. Gorbovitski, T. Rothamel, Y. Cheng, Y. Zhao, and J. Zhang. Core role-based access control: Efficient implementations by transformations. In *Proc. of the 2006 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 112–120, 2006.

Y.A. Liu, M. Gorbovitski, and S.D. Stoller. A language and framework for invariant-driven transformations. In *Proc. of the 8th Intl. Conf. on Generative Programming and Component Engineering*, pages 55–64, 2009.

P. Moor. Free Riding in BitTorrent and Countermeasures. *Master's Thesis, Distributed Computing Group, Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology Zurich*, 2006.

N. Nystrom, X. Qi, and A.C. Myers. J&: Nested intersection for scalable software composition. In *Proc. of the 21st Annual ACM SIGPLAN Conf. on Object Oriented Programming Systems, Languages, and Applications*, pages 21–36, 2006.

S. Romig. The OSU flow-tools package and Cisco NetFlow logs. In *Proc. of the 14th USENIX Conf. on System Administration*, page 304, 2000.

T. Rothamel and Y. A. Liu. Generating incremental implementations of object-set queries. In *Proc. of the 7th Intl. Conf. on Generative Programming and Component Engineering*, pages 55–66, 2008.

L. Ruy, M. Fluet, and U.A. Acar. Compiling self-adjusting programs with continuations. In *Proc. of the 13th ACM SIGPLAN Intl. Conf. on Functional Programming*, pages 321–334, 2008.

D. R. Smith. Requirement enforcement by transformation automata. In *Proc. of the 6th Workshop on Foundations of Aspect-Oriented Languages*, pages 5–14, 2007.

D. R. Smith. Aspects as invariants. *Automatic Program Development: A Tribute to Robert Paige*, pages 270–286, 2008.

SolarWinds. Orion NetFlow Traffic Analyzer. `http://www.solarwinds.com/Products/orion/nta/`, 2009.

M. Strembeck. Conflict checking of separation of duty constraints in RBAC — implementation experiences. In *Proc. of the 2004 Intl. Conf. on Software Engineering*, pages 224–229, 2004.

M. Ventuneac, T. Coffey, and I. Salomie. A policy-based security framework for web-enabled applications. In *Proc. of the 1st Intl. Symp. on Information and Communication Technologies*, pages 487–492, 2003.

E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. *Lecture Notes in Computer Science*, 3016:216–238, 2004.