Efficient Runtime Invariant Checking: A Framework and Case Study [*]

Michael Gorbovitski Tom Rothamel Yanhong A. Liu Scott D. Stoller
Computer Science Dept., State Univ. of New York at Stony Brook, Stony Brook, NY 11794
{mickg,rothamel,liu,stoller}@cs.stonybrook.edu

Abstract

This paper describes a general and powerful framework for efficient runtime invariant checking. The framework supports (1) declarative specification of arbitrary invariants using high-level queries, with easy use of information from any data in the execution, (2) powerful analysis and transformations for automatic generation of instrumentation for efficient incremental checking of invariants, and (3) convenient mechanisms for reporting errors, debugging, and taking preventive or remedial actions, as well as recording history data for use in queries. We demonstrate the advantages and effectiveness of the framework through implementations and case studies with abstract syntax tree transformations, authentication in a SMB client, and the BitTorrent peer-to-peer file distribution protocol.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Class invariants; D.2.4 [*Software Engineering*]: Software/Program Verification—Assertion checkers; D.2.13 [*Software Engineering*]: Reusable Software; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Invariants

General Terms Design, Languages, Performance, Verification

Keywords Alias analysis, Incrementalization, Program Transformation, Runtime Verification

1. Introduction

Program safety, security, and general correctness properties depend on all kinds of invariants holding during program execution. Even though static analysis can verify many invariants, many important invariants are still too difficult to verify automatically using static analysis. Therefore, it is critical to use dynamic techniques to check during program execution that these invariants hold. This is known as *runtime invariant checking*. It is challenging for at least three reasons:

1. invariants that relate information at multiple program points are difficult to specify and to verify at any one point in the execution,
2. the runtime overhead from invariant checking must be minimized, and
3. imminent violations of critical invariants must be detected before they occur, and appropriate actions taken in response.

This paper describes a general and powerful framework for efficient runtime invariant checking. The framework supports (1) declarative specification of arbitrary invariants using high-level queries, with easy use of information from any data in the execution, (2) powerful analysis and transformations for automatic generation of instrumentation for efficient incremental checking of invariants, and (3) convenient mechanisms for reporting errors, debugging, and taking preventive or remedial actions, as well as recording history data for use in queries. The transformations are built on InvTS rules [20], which describe how to incrementally maintain invariants.

We also describe a number of case studies that demonstrate the advantages of our framework and the effectiveness of our implementation. The implementation is for Python. The experiments include checking invariants about (1) abstract syntax tree (AST) transformations on programs of varying sizes between 400 and 16000 AST nodes, (2) Kerberos authentication used by a SMB client, and (3) a network protocol for distributing files in BitTorrent. All the invariants of interest can be expressed easily in our framework, and performance results show that our incremental checking scales well on large applications and complex invariants.

Much research has been done on runtime invariant checking, including a large variety of languages for specifying the invariants and methods for efficient instrumentation, as discussed in Section 5. To the best of our knowledge, no previous work both supports the generality of the kinds of invariants that our framework supports and achieves the efficiency that our implementation method achieves.

The rest of the paper is organized as follows. Section 2 gives an overview of our framework and describes the language for specifying invariants and actions. Section 3 describes analysis and transformations for incrementally checking the invariants. Section 4 presents our experiments. Section 5 discusses related work.

2. Framework

This section presents our framework for the specification of invariants and the actions to be taken when they are violated. Invariants are expressed as boolean conditions involving variables quantified over collections. Violations of an invariant correspond to tuples containing values of those variables for which the condition is false. We formulate runtime invariant checking as evaluating queries that return sets of such tuples. The basic form of an invariant checking rule in our framework is

[*] This work was supported by NSF under grants CCF-0613913, CNS-0509230, and CCR-0306399, and by ONR under grant N00014-07-1-0928.

WODA – Workshop on Dynamic Analysis, July 21, 2008
Copyright © 2008 ACM 978-1-60558-054-8/08/07…$5.00.

```
foreach ($v_1$ in $S_1$, ... , $v_k$ in $S_k$: condition):
    action
```

where S_1 through S_k are collections (sets, dicts and other collections that do not allow duplicates and that have constant-time membership tests). v_1 through v_k are quantified over sets S_1 through S_k, respectively. The set of tuples of values of v_1 through v_k such that *condition* holds is called the *query result*. *action* is a sequence of statements to be executed for each violation of the invariant, i.e., for each tuple in the query result.

For example, the following rule may be used to check that the `usage_count` field of each instance of the `File` class is non-negative.

```
foreach (o in extent(File): o.usage_count < 0):
   report("Error: File ", o, " has negative",
          " usage_count.")
   stop()
```

For every class C, `extent(`C`)` is a special set defined by our framework to contain the set of currently existing objects of type *C*. The `report` and `stop` functions are two functions in the subject programming language (Python): `report` takes any number of arguments and prints the concatenation of their string representations; `stop` stops the program and drops into a debugger, allowing the user to examine the state of the program at the point at which the invariant was violated.

While it is easy to see how to efficiently check simple invariants like the one above (by inserting checks at all assignments to the `usage_count` field), it becomes more difficult for even slightly more complex invariants. For example, consider a program that manipulates ASTs. We want to check that no node has an edge to itself. Assume that AST nodes are instances of the `Node` class, which has a `children` field. The invariant can be checked using the rule:

```
foreach (o in extent(Node): o in o.children):
   report("Error: ", o, " has a self-edge.")
   stop()
```

Checking this invariant efficiently is difficult, because aliasing implies that it can potentially be violated by any statement that adds an object to a collection, as in this scenario: `x=o.children; ...; x.add(o)`. Manually writing code to detect such bugs is tedious: one must intercept all calls to the `add` method of all instances of `set`, determine whether the target object equals the `children` field of some instance of `Node`, etc. In our framework, the user writes the simple rule above, and our system takes care of the rest, generating correct and efficient code for it.

Queries that involve multiple variables typically involve join conditions, which relate the values of the variables. For example, suppose the ASTs in the previous example should also satisfy the invariant that every node has at most one incoming edge. This can be checked using the rule:

```
foreach (n in extent(Node), m in extent(Node),
         c in extent(Node): c in n.children and
         c in m.children and n!=m):
   report("Error: ", c, "is a child of both ",
     m, " and ", n, ".")
   stop()
```

Again, it is easy to write this rule in our framework, but it is difficult to manually write code that efficiently checks this invariant at runtime, since this requires maintaining auxiliary data structures with information about edges, in addition to dealing with the aliasing issue discussed above.

Some invariants cannot be expressed using queries over extents and existing sets in the program. For example, consider a communication protocol. A query cannot refer to the set of all packets sent by the program, unless the program happens to maintain that set. It is not an extent, because packet objects are removed from the extent by garbage collection. To support such queries, our framework supports rules that add code throughout the program. This feature is similar to aspect-oriented programming, and it can be used to insert code that maintains additional sets.

```
foreach $(query)$ :
   $action$
$($de $($in $scope$ $(field|method)?$ $declaration)^*$ $)?$
$($at $update$
   $($if $condition)?$
   $($de $($in $scope$ $(field|method)?$ $declaration)^*$ $)?$
   $($do $($before $maint)?$ $($instead $maint)?$ $($after $maint)?$ $)?$
$)^*$
```

Figure 1. General form of an invariant checking rule.

The general form of an invariant checking rule is shown in Figure 1. The syntax of the new clauses is taken from InvTS [20], where they are used in rules that describe how to maintain invariants; this is why we use $update$ and $maint$ as suggestive names for the code patterns in the `at` and `do` clauses, but they are not limited to matching updates and specifying maintenance code. The `at` clause contains a code pattern $update$, which may contain subject-language code and meta-variables. Names of meta-variables start with "$". For each part of the code in the subject program that matches the *update* pattern in the `at` clause, if the *condition* in the `if` clause is satisfied, then the $declaration$s in the `de` (mnemonic for "declaration") clause are inserted into the program in the specified scope (while the `de` clause is usually used to declare and initialize variables, classes, or fields, it can be used to insert arbitrary code at a specified location) and the *maint* code in the `do` clause is inserted `before` or `after` the matched code, as specified, or, if `instead` is used in the `do` clause, the matched code is replaced with the code in the `do` clause. The condition in the `if` clause is built from standard logical connectives and functions defined for the subject language. For example, `class` $(expr)$ returns the class in which $expr$ appears, and `type` $(expr)$ returns the type of $expr$. In the `de` clause, *scope* can be `global` or the name of a class, method, or module.

Continuing the above example, the following rule could be used to check an invariant about packets that is expressed in terms of a set `$sent_packets` containing all sent packets (a specific example appears in Section 4). The meta-variable `$sent_packets` gets instantiated with a fresh program variable when the program is transformed.

```
foreach (...: ... $sent_packets ...):
   report("Error : ...")
   stop()
de in global:
   $sent_packets=set()
at $x.send($packet)
if extends(type($x),socket)
do before:
   global $sent_packets
   $sent_packets.add($packet)
```

3. Analysis and transformations

The straightforward way to implement the framework described above is to compute the result of every query from scratch at every program point. This is clearly correct, yet very slow, especially if the query involves large collections. A better way is to compute each query result at the program points that can update the result of the query. This is faster, yet still requires repeated evaluation of the query. A better approach is to efficiently maintain (i.e., update) the result of the query whenever a collection or object the query depends on changes.

This requires two steps: (1) generating maintenance code that properly maintains the query results in the face of updates to the data the query depends on, and, (2) applying the maintenance code at all places where the query result might change. The rest of the section uses "set" instead of "collection" as the method applies (with very minor modifications) to any collection that contains objects, does not allow duplicates, and has constant-time membership testing.

Step 1 is accomplished by compiling the query into an InvTS rule [20], which then transforms the subject program so that it incrementally maintains the query result. InvTS (the Invariant-Driven Transformation System) is a program transformation system that is geared towards source-to-source transformations that maintain invariants.

Step 2 is performed by InvTS itself. To maintain the result of a query, InvTS inserts the maintenance code from step 1 at every location that updates the variables the query depends on. The straightforward way is to insert maintenance code at every statement in the program, preceded by a runtime check of whether the statement actually updates the data the query depends on. This slows down the transformed program even when no such updates occur, due to the evaluation of the runtime check at every statement. InvTS uses control-flow, data-flow, type, and alias information to evaluate as many of these checks as possible at compile time, to reduce the runtime overhead of maintaining the query result.

Generating maintenance code. As InvTS alone cannot generate the code to maintain a query result, we give a method that, for a class of queries, generates maintenance code (in the form of InvTS `at/if/de/do` clauses) that incrementally maintains the result of these queries.

We generate efficient maintenance code for queries of the form (v_1 `in` S_1, ..., v_k `in` S_k: *condition*), where *condition* is a conjunction and each conjunct is either (1) a join condition of the form e_1 *op* e_2, where *op* is `==`, `!=`, `in`, or `not in`, and e_i is v or $v.f$, where v is a variable and f is a field, or (2) a boolean expression whose value depends only on the objects bound to $v_1, \ldots, v_k$, the fields of these objects, and immutable objects.

Three kinds of updates can affect the result of a query: adding an object to a set, removing an object from a set, and changing the value of a field on an object. We decompose more complicated updates into these simple updates. We further simplify the problem by replacing field updates (for both scalar and set fields) with code that removes an object from all sets containing it, updates the field, and re-adds it to all sets. This transformation requires maintaining an auxiliary map from each object to the sets containing it.

With this simplification, the query result can increase only when an object is added to any of the sets $S_1, \ldots, S_k$, and the query result can decrease only when objects are removed from these sets. Since the action is executed only when the result set increases, this means that we only need to handle the addition case appropriately to update the query result. However, during removal we may need to update auxiliary maps.

Handling element addition. To handle addition of an object to a set, we run the query with the corresponding `v` variable bound to the object being added. We then generate statements corresponding to each of the clauses (enumeration, predicate, and join) in the query. The code is generated in the following order:

1. For a predicate with all variables bound, an if-statement checking the predicate is generated.
2. For an enumeration of the form v `in` S where v and S are both bound, an if-statement that performs a membership test is generated.
3. For a join condition with both variables bound, an if-statement that checks whether the join condition is satisfied is generated.
4. For an equality or set-membership join with exactly one variable bound, a for-statement that iterates over the entry corresponding to the bound variable in a hash-join map is generated.
5. For an enumeration where only S is unbound, a for-statement that iterates over the elements of S is generated.

If a clause does not match one of the conditions in this list, then it cannot be generated yet. Each generated for-statement binds a variable, which can cause statements to become generable or to rise in priority. As all variables can be bound through the for statement, eventually all clauses will be generated. The generated InvTS code has the form of additional `at`, `if`, `de`, and `do` clauses that, `at` each element addition, `do` the above-described maintenance.

Handling joins. For each join, we maintain a hashmap, which we call a hash-join auxiliary map. For example, for the join `v1.parent==v2.name`, if `v1` is bound, and `v2` iterates over `S2`, we introduce a hashmap with domain `S2` that maps `o.name` to `o`. Maintaining these mappings requires the generation of additional code which must be run in response to the addition and removal of elements of `S2` and changes to `o.name`. This code must be run before the maintenance code that handles element addition. Thus, either new `at/if/de/do` clauses are created, or existing ones are modified so that the new maintenance code is prepended to the appropriate `do` clauses.

Auxiliary clauses. The `at`, `if`, `de` and `do` clauses have the same syntax and meaning as in InvTS. Thus they are copied into the InvTS rule being generated.

Type analysis. Our system uses static type analysis to reduce the number of runtime checks. If a variable of a known type is being updated, and variables (or fields) of this type are not used in the query, then the update cannot affect the result of the query, and the corresponding runtime check can be eliminated.

Our type system expands on Python's type system by making it more precise. We introduce types that represent constants, abstract values such as lists of constant lengths, and unions of two or more types. This higher precision, plus static analysis of the types, in contrast to Python's dynamic type analysis, allows InvTS to evaluate a large number of checks statically. In our experiments, this reduces overhead by a factor of two or more in most cases, based on Table 1 in Section 4. For example, for "BitTorrent - No duplicate data", this reduces CPU usage from 3.9% with type analysis disabled, to 3.3% with type analysis enabled; given the CPU usage of 2.7% for the program running without any checks, this reduces the overhead of invariant verification from 44% to 22%. For the other BitTorrent experiment, the overhead is reduced from 125% to 28%.

Alias analysis. InvTS also uses alias analysis to reduce the number of runtime checks, as an update to a variable that is not aliased to a variable in the query cannot affect the query result. Clearly, more precise alias analysis allows more runtime checks to be eliminated. We use a flow-sensitive interprocedural may-alias algorithm, in contrast to simpler but less precise flow-insensitive algorithms such as Andersen's.

The alias analysis algorithm we use is based on the intraprocedural, flow-sensitive may-alias analysis by Goyal [10]. Goyal's algorithm is intraprocedural, works on C, and has a running time of O(n^3). Thus, it had to be extended to handle Python, and to work interprocedurally. This resulted in a worst-vase complexity of O(n^4), although in practice, for all programs we analyzed, the running time increased quadratically with the size of the program. In our experiment "InvTS - No own child", alias analysis reduces the overhead from 100% (Overhead of "No Alias Analysis" compared to "No Check") to 62% (Overhead of "Incremental" compared to "No Check"), as shown in Table 1.

4. Experiments

To demonstrate that our technique can efficiently verify invariants, we have applied it to invariants from multiple domains: abstract syntax tree transformations, authentication, and a file distribution protocol. For each invariant, we compare the performance of the program without any invariant checking; with invariants being checked incrementally using the method described in this paper; and with invariants checked in a non-incremental manner by re-evaluating the query from scratch each time an update occurs.

All experiments were performed using Python 2.5.1 on Windows Vista, running on a Core 2 Duo (Q6600@3.0GHz) machine with 8GB of memory, of which 6GB were free.

4.1 AST transformations

An abstract syntax tree (AST) should satisfy several invariants. For our first two experiments, we check that no AST node is its own child, and that each AST node is the child of at most one parent.

For these experiments, we apply InvTS to itself to create checked-InvTS, a version of InvTS that checks to ensure that program transformations do not violate the AST invariants. Checked-InvTS is then run with a rule-set that transforms subject programs into static single-assignment (SSA) form. Note that in this case, we are checking the correctness of checked-InvTS, rather than the programs it is applied to.

Not own child. Recall from Section 2 that the following rule detects violations of the invariant "a node is not a child of itself".

```
foreach (o in extent(Node): o in o.children):
   report("Error: ", o, " has a self-edge.")
   stop()
```

Figure 2 shows that checking this invariant cause a constant factor slowdown. The overhead is close to 70%. About half of this overhead is the cost of maintaining extents, while the other half is the cost of maintaining invariants.

We do not give the running time of the non-incremental instrumentation, as not even the smallest experiment was able to complete in the time limit of 20 minutes. Since the query is run each time an AST node is created or updated, the non-incremental version incurs an asymptotic slowdown. Incremental instrumentation eliminates this asymptotic penalty, rendering invariant checking practical.

No shared child. In an AST, no two parents may refer to the same child. The following rule checks for violations of this invariant:

```
foreach (n in extent(Node), m in extent(Node),
         c in extent(Node): c in n.children and
         c in m.children and n!=m):
   report("Error: ", c, "is a child of both ",
     m, " and ", n, ".")
   stop()
```

As this invariant contains multiple join conditions (`c in m.children`, `c in n.children`, `n!=m`), hash-join maps are used to evaluate it efficiently.

Figure 2 shows that incrementally checking this invariant increases the running time by less than 95%. In contrast, the non-incremental instrumentation would be cubic in the number of nodes currently alive in the program, as it iterates over three extents of nodes. This leads us to the estimate that, in the best case, the non-incrementally instrumented program is O($\#node^3$) worse than the uninstrumented one. It is not a surprise that all experiments with non-incremental instrumentation timed out. When we manually introduced a bug that assigned the same child to multiple parents, checked-InvTS detected the violation.

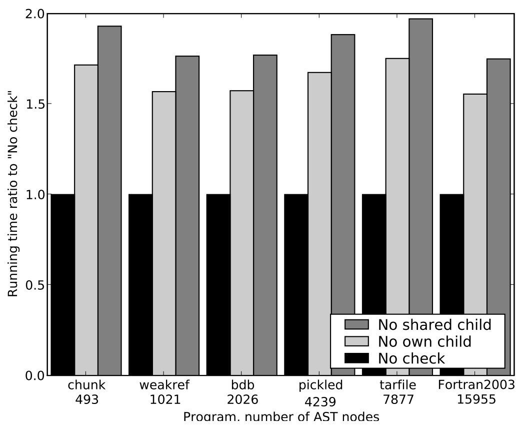

Figure 2. Running times of InvTS normalized to the running time of the non-instrumented version.

Overall, these experiments show that verifying invariants at runtime can be efficient (with overhead smaller than 95%) for even complex queries that involve multiple joins and membership tests. We also see that when joins used by the query have a high selectivity, as these do, the running time of the instrumented program is not very dependent on the query, but more so on the number of classes for which we maintain extents.

4.2 Authentication

We performed two experiments involving the Kerberos authentication used by pysmb, a SMB client written in Python. The first checks that all packets sent are authenticated; the second checks that authentication does not occur more frequently than necessary.

Require valid ticket. Our first experiment checks that we do not send packets to hosts that have an invalid Kerberos ticket associated with them. This invariant needs to remain true until the packet is actually sent. To find violations of it, we keep a set of packets being sent, and report an error if a packet in the set is associated with an invalid ticket.

```
foreach (sp in $sending_packets,
         kt in extent(KerberosTicket):
         kt.invalid and kt.ip==sp.target_ip ):
   report("Sending ", sp, " with invalid ticket!")
   stop()
de in global:
   $sending_packets=set()
at $x.send($p):
if subclass(type($x),asyncore.dispatcher):
de in class type($x) in function handle_write($arg):
   if $arg in $sending_packets:
      $sending_packets.remove($arg)
do after:
   if $p not in $sending_packets:
      $sending_packets.append($p)
```

This rule tracks all sends of data over asynchronous sockets, and stops the program when a packet was sent to a server with an invalid

Kerberos ticket. The `de` and `do` clauses work in the following manner: When a `send` method call is encountered, the packet being sent is added to the `$sending_packets` queue. It is removed from there once the packet is actually sent, which may not be necessarily immediate. This is detected by intercepting the `handle_write` callback in the class subclassing `asyncore.dispatcher`. This callback is called by Python when a packet is actually sent out over the given socket.

When we ran this on pysmb, while transferring a 10GB file over a 100Mbit/sec connection, the average CPU load increased from 3.6% to 11.7%. The throughput remained the same, because the program was IO-bound in both cases. The increase is due to the join and the fact that many Kerberos tickets may match an IP address. A straightforward implementation increased CPU usage to 97%, and reduced the throughput of the program by 73%, as pysmb became CPU-bound. The times taken by the program to transfer the file were 1302 seconds for the uninstrumented version, 1351 seconds for the incrementally instrumented version, and 6321 seconds for the non-incrementally instrumented version.

Repeated authentication. It is inefficient for a program to request tickets from the Kerberos server long before the currently valid ticket times out. Thus, a useful invariant to check is that a successful authentication is not repeated until the resultant ticket is about to time out. A ticket times out when there was no activity relating to that ticket for 300 seconds, i.e., no data was sent to the host the ticket was issued for for the last 300 seconds. Thus the invariant is: there are no two valid tickets such that they are both referring to the same host, are both valid, and are much less than `timeout` (i.e., 300 seconds) apart. We define much less as 10 seconds less, as the MIT Kerberos client requests a new ticket 10 seconds before the current one times out. To verify this invariant, we need to keep track of Kerberos tickets and of SMB activity.

The invariant is expressed using a nested query, with the inner query computing the latest packet sent to a given host, and the outer query doing a join on all pairs of currently existing Kerberos tickets. The max aggregate is maintained using a heap.

```
foreach (k_old in extent(KerberosTicket) ,
         k_new in extent(KerberosTicket):
         k_old.valid and k_new.valid and
         k_old.issue_time<k_new.issue_time and
         k_old.ip==k_new.ip and
         k_new.issue_time-max([p.time
            for p in $sent_packets
            if p.target_ip==k_new.ip and
               p.time < k_new.issue_time])
         < 300-10):
   report ("Reauthenticated to host ", k_new.ip )
   stop()
de in global:
   $sent_packets=set()
at $x.send($p)
if type($x)==asyncore.dispatcher
do after:
   $sent_packets.add($p)
```

When run on pysmb, while transferring a 10GB file over a 100Mbit/sec connection, the average CPU load increased from 3.6% to 17.9%, mainly due to the need to maintain a heap per IP address, and an additional join over the previous example. Using specific domain knowledge, the heap could be avoided: we could just keep track of the latest packet sent to each IP address. This works because time is monotonic. A rule modified in such a way is less easily adapted towards other uses, though. Note that even with the maintenance of the heap, the instrumented program is still IO bound, not CPU bound. Checking invariants in a non-incremental manner makes it CPU bound: it results in a 96.9% CPU load, and the running time increases from 1302 to 8750 seconds.

The pysmb examples show that instrumenting complex programs in ways not anticipated by their creators is easily done with our framework due to the ability to specify complex program transformations, such as maintaining the set of sent packets, or the set of packets waiting to be sent. It also demonstrates that complex conditions, including nested queries, are supported by this framework, and their use does not cause excessive overhead.

4.3 File distribution protocol

BitTorrent (http://download.bittorrent.com/dl/) is a peer-to-peer file distribution 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 very large numbers of downloaders with only a modest increase in its load. Each peer downloads chunks of a file from (likely different) peers, and then reassembles the original file from the chunks. The BitTorrent protocol is relatively complex, so we use our method to instrument an implementation and check it for potential errors.

No duplicate data. Receiving the same piece of data from two sources too often may mean that the client is using bandwidth inefficiently. We check for this using a rule that detects when the same data is received from two or more distinct sources (identified by IP address), and logs the event without stopping the program. The log could be analyzed later to determine whether the duplication is due to a bug or misconfiguration.

```
foreach (p1 in $in_queue, p2 in $in_queue:
         p1.source_ip!=p2.source_ip and
         p1.payload==p2.payload):
   report("Receiving same data from peers ",
     p1.source_ip, " and ", p2.source_ip)
de in global:
   # A queue of incoming packets.
   # It supports O(1) membership tests,
   # holding at most 100000 packets
   $in_queue=queue(max_length=100000)
at $x.type=$s
if $s=="incoming" and type($x)==Packet
do after:
   if $x not in $in_queue:
      $in_queue.append($x)
```

Experiments involved receiving a 10GB file from 30 peers, over a 100Mbit/sec connection. We measured CPU load to determine the impact of the runtime checking. The average CPU load increased from 28.3% for the original program to 36.1% for the instrumented program. The small increase is due to the high selectivity of the `p1.payload==p2.payload` join condition. Just like with pysmb, both versions of the program are IO-bound.

No packet modification in transit. To verify that the correct data is being sent between peers, we check the following invariant: a packet sent from one peer must be received by another peer without a change in the payload.

We check this invariant by creating a server to which peers send summaries of the packets they send and receive. These packets are put into a set on the server. We write a query that detects when packets of the same chunk have a different payload, by comparing the MD5 hashes of the payloads.

The server maintains a set `rec_set` containing all packets sent and received by BitTorrent peers. The following rule checks the invariant:

```
foreach ($from in self.rec_set, $to in self.rec_set:
```

	No Check	Incremental	No Type Analysis	No Alias Analysis	Non-Incremental
pysmb - Require valid ticket	3.6% (1302s)	11.7% (1351s)	19.7% (1819s)	14.1% (1601s)	97.3% (6321s)
pysmb - Repeated authentication	3.6% (1302s)	17.9% (1535s)	31.7% (2011s)	23.3% (1943s)	96.9% (8750s)
BitTorrent - No duplicate data	28.3% (1771s)	36.1% (1779s)	63.8% (1790s)	36.3% (1830s)	99.8% (3210s)
BitTorrent - No packet modification	2.7% (1783s)	3.3% (1687s)	3.9% (1763s)	3.4% (1805s)	93.1% (1801s)
InvTS - No shared child	13s	25s	349s	25s	>1200s
InvTS - No own child	13s	21s	312s	26s	>1200s

Table 1. CPU utilization (if IO-bound) and wall time taken for experiments under differing optimizations.

```
        $from!=$to and $from.source!=None and
        $from.target!=None and
        $from.source==$to.source and
        $from.target==$to.target and
        $from.chunk==$to.chunk and
        $from.chunk!=None and
        $from.sent and $to.received and
        $from.md5!=$to.md5 and $from.md5!=None):
   report ("Packet sent from ", $from.source,
     " to ", $from.target, " changed in transit!")
   stop()
```

We use two InvTS rules to modify the BitTorrent program to send the information needed for invariant verification to the server. The rules state that a socket should be opened to the server once per program, and that anytime a packet is written to any socket, or read from any socket, the packet (minus the body) should be sent to the server. The rule for handling `send` is the same as the rule below for handling `receive`, with `receive` replaced with `send`.

```
at $x.receive($p)
if type($x)==asyncore.dispatcher
de in global:
   import socket
   #Open a socket to server on 192.168.17.46:636
   $check_socket=socket.open_udp(192.168.17.46,636)
   in global in function(myreceive(socket,packet)):
      global $check_socket
      # For efficiency, do not sent the payload
      $body=packet.body
      $arg.body=None
      $check_socket.send(packet)
      packet.body=$body
do instead:
   myreceive($x, $p)
```

After applying the query and rules to the BitTorrent client and our server, we benchmarked the CPU utilization of the clients and the server (which were running on the same computer). With 5 BitTorrent clients and the server running, the CPU utilization increased from 73 to 78 percent. When the clients were measured in isolation, the CPU utilization of a single client (with the other 4 clients and the server running on another system) was 11%, vs. 10% for the untransformed client. The server, when run on the test machine (with the 5 clients running on a different machine) utilized 3.3% of the CPU with the instrumentation enabled, versus 2.7% with no instrumentation.

On a reliable connection we found no problems. When we simulated a bad connection by randomly injecting changes into some packets, we found the errors before the BitTorrent error detection algorithm, which operates on bigger chunks.

Effect of optimizations. Table 1 shows the CPU utilizations and running times of the pysmb and BitTorrent examples under different implementation options. It is easy to see that the non-incremental implementation is far worse than any other version. Disabling type or alias analysis also produces a noticeable slowdown.

5. Related work and conclusion

This paper touches two areas: runtime invariant verification [8], and incremental query result maintenance.

There are several systems for runtime checking of temporal properties. These include Java-MaC [15], JPAX [13] , JNuke [1], and EAGLE [4]. These systems express the properties in a linear temporal logic (LTL) or a related rule languages.

Our system does not support writing invariants in LTL, although, as our system supports comprehensions, extents, and joins, a subset of LTL can be emulated. The pysmb example does so by maintaining history and specifying queries over it. While this may incur a performance penalty compared to systems specifically designed to test LTL-based invariants, it is not a very significant performance penalty (As seen in Section 4, the overhead is consistently under 100%).

The category into which our system fits best is tools that use a side-effect free subset of their subject language, extended with various operators such as quantifiers or set operations, to specify invariants. Such invariant specification languages include JML [19], Spec# [3], and Jahob [17]. For JML and Spec#, there are tools that allow the user to combine/compile an invariant and a subject program into a compiled program that, at runtime, checks whether the specified invariant holds. These tools include Boogie [2] for Spec# and jmlc [7], jass [5], jmle [16], and DITTO [24] for JML. A runtime verifier for Jahob is under development [28].

Spec# does not support comprehensions[28]; or extents. As such, it cannot easily encode the invariants we wish to verify. JML supports set comprehensions, quantifiers, and other features. It does not natively support extents [18]. Jahob supports both comprehensions and extents (as a subset of the `AliveVariables` set). The language presented in this paper supports both set comprehensions and extents. It is worth noting that support for extents is difficult to emulate without support for liveness testing, because garbage collection must be taken into account.

The JML compilers jmlc, jmle, and jass all support a large subset of JML, including comprehensions. But, they evaluate comprehensions in a straightforward manner, by recomputing them whenever they are encountered. In contrast, our system incrementally maintains the value of set comprehensions. DITTO provides incremental maintenance of some JML expressions, but it does not incrementally maintain set comprehensions [24].

JQL [27] extends Java to support both comprehensions and extents, to support querying over collections. Recent work on JQL adds incremental maintenance of JQL queries in the face of updates to the data they depend on. The fact that our system is designed with only invariant verification in mind allows us to more efficiently maintain invariants. For example, it is easier for us to handle removal of elements from the sets that the query depends on. We support a marginally larger set of conditions on queries: we can incrementally maintain query results for queries that contain a condition of the form `a in b.f`. Also, the `at` and `de` clauses allow us to do program transformations that maintain datastructures that would be unavailable to a pure query language, such as a set of all previously sent packets.

Potanin et al. [23] query snapshots of object graphs, but perform the queries non-incrementally. PQL [21] queries over past states of the program, but not over extents. It uses BDDs to compute query results, but not incrementally.

Aspect-oriented programming can be used to check invariants. The user can directly write pointcuts and advice to check an invariant and take appropriate action on violations [14, 25]. As we have shown in Section 3, for even moderately complex invariants this is tedious and error-prone. Alternatively, the user can write a tool that generates pointcuts and advice from a specification [26, 6, 9]. These tools are task-specific, so the user will likely have to write such a tool for his particular task. This is non-trivial, especially if the user wants to create a tool that will generate advice that incrementally verifies invariants. Our system lets the programmer avoid manually writing pointcuts and advices that incrementally maintain invariants, as well as absolving him of the responsibility of writing a system that generates such pointcuts and advices. Instead, it lets him concentrate on the task of specifying invariants.

There is a large amount of work on incremental maintenance of invariants, e.g, [11, 22, 12, 20, 24]. From these, especially relevant to this paper is our system InvTS [20], which applies rules that incrementally maintain query results. We use InvTS to apply rules generated from queries in debugging rules. The advantage of InvTS is its utilization of static analysis to reduce runtime overhead, as described in Section 3.

Using our framework for other languages such as Java and C requires implementing the framework as described in Section 3, including in particular implementing the type and alias analysis algorithms for the desired language. As a proof of concept, we have extended InvTS to transform GCC C, where we implemented our interprocedural alias analysis and used GCC's built-in type analysis. Future work includes refinements and experiments for our InvTS implementation for GCC C.

References

[1] C. Artho, V. Schuppan, A. Biere, P. Eugster, M. Baur, and B. Zweimüller. JNuke: Efficient dynamic analysis for Java. *Lecture Notes in Computer Science*, 3114:462–465, 2004.

[2] M. Barnett, B. Chang, R. DeLine, B. Jacobs, and K. Leino. Boogie: A modular reusable verifier for object-oriented programs. *Proc. of the 4th Intl. Symp. on Formal Methods for Components and Objects*, pages 364–387, 2006.

[3] M. Barnett, R. DeLine, M. Fahndrich, K. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[4] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. *Proc. of the 5th Intl. Conf. on Verification, Model Checking and Abstract Interpretation*, pages 44–57, 2004.

[5] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass Java with assertions. *Electronic Notes in Theoretical Computer Science*, 55(2):103–117, 2001.

[6] F. Chen and G. Roşu. Mop: an efficient and generic runtime verification framework. *Proc. of the 22nd annual ACM SIGPLAN Conf. on Object Oriented Programming Systems and Applications*, pages 569–588, 2007.

[7] Y. Cheon. *A Runtime Assertion Checker for the Java Modeling Language*. PhD thesis, Iowa State University, 2003.

[8] L. Clarke and D. Rosenblum. A historical perspective on runtime assertion checking in software development. *ACM SIGSOFT Software Engineering Notes*, 31(3):25–37, 2006.

[9] T. Gibbs and B. Malloy. Weaving aspects into C++ applications for validation of temporal invariants. *Proc. of the 7th European Conf. on Software Maintenance and Reengineering*, pages 249–258, 2003.

[10] D. Goyal. Transformational derivation of an improved alias analysis algorithm. *Higher-Order and Symbolic Computation*, 18(1/2), Feb. 2005.

[11] D. Gries. *The Science of Programming*. Springer, 1981.

[12] A. Gupta, I. Mumick, and V. Subrahmanian. Maintaining views incrementally. *Proc. of the 1993 ACM SIGMOD Intl. Conf. on Management of Data*, pages 157–166, 1993.

[13] K. Havelund and G. Roşu. An Overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design*, 24(2):189–215, 2004.

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[15] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a run-time assurance approach for Java programs. *Formal Methods in System Design*, 24(2):129–155, 2004.

[16] B. Krause and T. Wahls. jmle: a tool for executing JML specifications via constraint programming. *Lecture Notes in Computer Science*, 4346:293–296, 2007.

[17] V. Kuncak and M. Rinard. An overview of the Jahob analysis system: project goals and current status. *20th Intl. Parallel and Distributed Processing Symp.*, pages 8–16, 2006.

[18] G. Leavens, A. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

[19] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of jml accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1-3):185–208, 2005.

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

[21] M. Martin, B. Livshits, and M. Lam. Finding application errors and security flaws using PQL: a program query language. *ACM SIGPLAN Notices*, 40(10):365–383, 2005.

[22] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, 1982.

[23] A. Potanin, J. Noble, and R. Biddle. Snapshot query-based debugging. *Proc. of Australian Software Engineering Conf.*, pages 251–259, 2004.

[24] A. Shankar and R. Bodík. DITTO: automatic incrementalization of data structure invariant checks (in Java). *Proc. of the 2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 310–319, 2007.

[25] F. Steimann. The paradoxical success of aspect-oriented programming. *Proc. of the 21st Annual ACM SIGPLAN Conf. on Object-oriented Programming Languages, Systems, and Applications*, pages 481–497, 2006.

[26] V. Stolz and E. Bodden. Temporal assertions using AspectJ. *Electronic Notes in Theoretical Computer Science*, 144(4):109–124, 2006.

[27] D. Willis, D. Pearce, and J. Noble. Efficient object querying for Java. *Proc. of the European Conf. on Object-Oriented Programming*, pages 28–49, 2006.

[28] K. Zee, V. Kuncak, M. Taylor, and M. Rinard. *Lecture Notes in Computer Science*, 4839:202–213, 2007.