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 implementation of the BitTorrent peer-to-peer file sharing communication protocol.

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 should be minimized, and
3. imminent violations of critical invariants should be detected before they occur, and appropriate actions should be 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 to use in queries. The transformations are built on InvTS rules [18], 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 trees (AST) transformations on programs varying sizes between 400 and 16000 AST nodes, (2) Kerberos authentication used by a SMB client written in Python, and (3) implementation of network protocols for distributing files in BitTorrent. In these experiments, 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, including methods for incremental checking for certain kinds of invariants, 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 experiments that show the effectiveness and efficiency of our framework and implementation. Section 5 discusses related work.

2. Framework

This section presents our framework for specification of invariants and 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. Therefore, we formulate runtime invariant checking as evaluating queries that return such sets of tuples. Thus, the basic form of an invariant checking rule in our framework is

`foreach` (*v1* `in` *S1*, *v2* `in` *S2*, ... : *condition*):
    action

where *S1*, *S2*, ..., are collections (sets, lists, etc.). The set of tuples of values of *v1*, *v2*, ..., such that *condition* holds is called the *query result*. *action* is a statement to be executed for each violation of the invariant, i.e., for each tuple in the query result.

For example, the following rule could 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 in this example are two functions in the subject programming language (Python): `report` takes any number of arguments and prints the concatenation of their string representations, and `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 even for slightly more complex invariants. For example, consider a program that manipulates ASTs, and we want to check that no node has an edge to itself. Assume that AST nodes are instances of an `ASTNode` class that declares a `children` field. The invariant can be checked using the rule:

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

Checking this invariant efficiently is more 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 a 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 graphs in the previous example should also satisfy the invariant that every node has at most one incoming edge. This can be checked using a rule such as:

```
foreach (n in extent(ASTNode), m in extent(ASTNode),
         c in extent(ASTNode) : 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.

The general form of an invariant checking rule is shown in Figure 1. The meaning of the new clauses is as follows. The syntax

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

Figure 1. General form of an invariant checking rule.

of these clauses is taken from InvTS [18], 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. Meta-variables are denoted by prefixing their name with "$". For each part of the code in the subject program that matches the pattern in the `at` clause, if the *condition* in the `if` clause is satisfied, then the declarations in the `de` (mnemonic for "declaration") clause are inserted in the program in the specified scope, 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. In the `if` clause, the condition 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, package, or file.

Continuing the above example, the following rule could be used to check an invariant about packets that is expressed in terms of a set `$sent_packets` containing all sent packets (a specific example appears in Section 4). Note that the meta-variable `$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, for every query, to ask at every program point "What is the result of the query?". This is clearly correct, yet very slow, especially if the size of the sets queried over iterated over is large. A better way is to only ask at the program points that could possibly update the result of the query. This is clearly faster, yet still causes us to repeatedly reevaluate the query. A better approach is to efficiently maintain the result of the query whenever a set or object the query depends on changes.

Doing this requires two steps: (1) generating the maintenance code that will properly maintain the results of the query in the face of updates to the data the query depends on, and, (2) applying the maintenance code at all places that change the query's results.

Step 1 is accomplished by compiling the query into an InvTS rule [18], which then transforms the subject program so that it maintains the query's value. InvTS (the Invariant-Driven Transformation System) is a program transformation system that is geared towards source-to-source transformations that maintain invariants of the following type: a variable is equal to the result of a query with respect to all possible updates to the sets and fields the query depends on.

Step 2 is performed by InvTS itself. To maintain the results of a query, InvTS inserts user-provided maintenance code 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 that verifies that the statement actually updates the data the query depends on. This slows down the transformed program even when no updates occur, due to the evaluation of the runtime check at every statement in the program. InvTS uses control-flow, data-flow, type, and alias information to evaluate as many of these checks as possible at compile-time, thus reducing the overhead of maintaining the query result.

Generating maintenance code. As InvTS alone cannot derive the actual code to maintain the value of the query, we give a method that, for a class of queries, generates maintenance code that incrementally maintains the result of these queries. This method is a subset of the method given in [20].

We generate efficient incremental maintenance code for queries of the form `v1 in S1, v2 in S2, ... | condition`. The condition is a conjunction of predicates and joins. Predicates may only depend on the variables `v1`, `v2`, etc, and their immediate fields (`v1.a` is allowed, but `v1.a.b` is not), while joins must be of one of the following forms: `v1==v2`, `v1!=v2`, `v1==v2.field`, `v1!=v2.field`, `v1.field1==v2.field2`, `v1.field1!=v2.field`, `v1.field in v2.field`, and `v1 not in v2.field`. `S1` and `S2` must have constant-time membership testing.

There are three kinds of updates that can affect the result of these queries: the addition of an object to a collection, the removal of objects from a collection, 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 collection 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.

Finally, we note that the query result only increases when objects are added to `S` sets (`S1`, `S2`) , and the query result only decreases when objects are removed from the `S` sets. Since we only execute the action body when the result set increases, this means that we only need to handle the set addition case. However, note that during set removal we may update auxiliary maps.

Handling element addition. To handle addition of an object to a `S` 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 (iteration, predicate, and join) in the query. The code is generated in the following order:

1. Predicates with all variables bound, generates an if-statement evaluating the predicate.
2. Iteration with both variables bound, generates an if-statement with a membership test.
3. Joins with both variables bound, generates an if-statement that tests membership in a hash-join map.
4. Equality and set-membership joins with one variable bound, generates a for-statement that iterates over the appropriate entry in a hash-join map.
5. Iteration with only the set variable bound, generates a for-statement that iterates over the set.

If a clause does not match one of the conditions in this list then it cannot be generated. Each generated for-statement binds a variable, which can cause statements to become generatable or to rise in priority. As all variables can be bound through iteration, eventually all clauses will be generated.

Handing joins. For each join, we maintain a hash-join auxiliary map. For example, if we have the join `v1.parent==v2.name`, `v1` is bound, and `v2` iterates over `S2`, for each object `o` in `S2` there is a mapping from `o.name` to `o`. Maintaining these mappings requires the generation of additional auxiliary code, which must be run before the maintenance code given above. This code may be run in response to addition and removal.

Auxiliary clauses. As auxiliary clauses have the same syntax as InvTS, all auxiliary clauses (if, do, de, at) are copied into the InvTS rule being generated.

Type analysis. Static type analysis can be used to reduce the number of runtime checks, because if a variable of a given 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 does not need to be performed.

Our type system expands on Python's type system by making it more precise. We introduce types that represent constants, lists of known length, lists of known type, lists of known content, empty vs. non-empty strings, positive and negative numbers (integers, floats, etc.), types which are the union of two or more types, etc. This higher precision, coupled with making the type analysis static (Python only provides dynamic type analysis), allows InvTS to evaluate a large number of runtime checks statically. From our experiments, the overhead reduction due to type analysis is from 30% to 100%, as seen in Table 1.

Alias analysis. Alias analysis is also used by InvTS to reduce the number of runtime checks, as an update to a variable that is not aliased to a variable or field in the query cannot update it. Clearly, the more conservative the alias analysis, the less runtime checks can be removed. That is why we use a flow-sensitive interprocedural may-alias algorithm, in contrast to the more conservative, but easier-to-implement flow-insensitive algorithms such as Andersen's.

The alias analysis algorithm we use has time complexity of $O(n^4)$, and is based on the intraprocedural, flow-sensitive may-alias analysis by Goyal [8]. 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 an increased asymptotic complexity of $O(n^4)$, although in practice, for all programs we have analyzed, we have always observed the running time increasing quadratically with the size of the program. From our experiments, the overhead reduction due to alias analysis is from 30% to 50%, as seen 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 protocol implementation. For each invariant, we compare the performance of the program without any invariant checking; with invariants being checked incrementally using 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 on Windows Vista, running on a Core 2 Duo (Q6600@3.0GHz) machine with 8GB of memory, of which 6GB were free. For all examples, Python 2.5.1 was used.

4.1 AST transformations

For an abstract syntax tree (AST) to be correct, there are a number of invariants it must satisfy. 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. If an AST transformation system violates these invariants, it is incorrect.

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 single-assignment 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. In an abstract syntax tree, a node may not have itself as a child. We haver written a rule that reports cases where this invariant is violated, then stops the program so that the programmer can investigate:

```
foreach (o in extent(ASTNode) :  o in o.children ) :
   report(o, " is a child of itself!")
   stop()
```

Figure 2 shows that the running time of checked-InvTS when verifying the not own child invariant is within a constant factor of the running time of the uninstrumented version. The overhead incurred by the instrumentation is close to 70%. About half of this overhead was caused by the overhead required to maintain extents, while the other half was 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, we expect the non-incremental version to incur an asymptotic slowdown. Incremental instrumentation eliminates this 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(ASTNode), m in extent(ASTNode),
         c in extent(ASTNode) : c in n.children and
         c in m.children and n!=m ):
   report("Child ", c, "is a child of both ",
     m, " and ", n, "!")
   stop()
```

As this invariant accesses multiple extents, hash joins are required to evaluate it efficiently.

Figure 2 shows that the running time of the incrementally instrumented program remains less than double that of the uninstrumented version. 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. This, coupled with the fact that the smallest program we look at is over 400 AST nodes, accounts for the fact that all experiments with non-incremental instrumentation timed out. When we manually introduced a bug that would assign the same child to multiple parents, checked-InvTS would stop when run, and give us a debugging shell.

Overall, these experiments show that verifying invariants at runtime can be efficient (with overhead smaller then 80%) for even complex queries that involve multiple joins and membership tests.

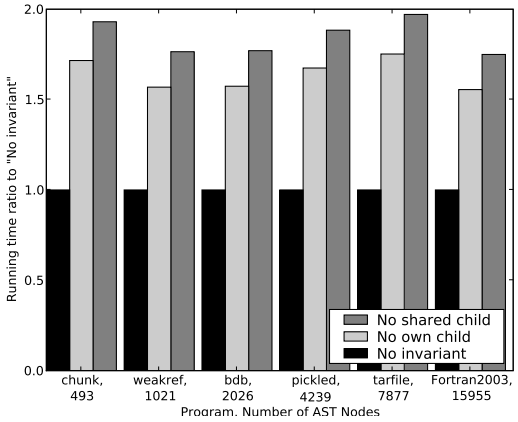

Figure 2. Running times of InvTS normalized to the running time of the non-instrumented version.

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

Another important class of invariants are those required to confirm that authentication protocols are being performed correctly. We performed two experiments involving the Kerberos authentication used by pysmb, a SMB client written in Python. These check that all packets sent are authenticated, and that authentication does not occur more frequently than is necessary.

Require valid ticket. For our first experiment, we want to verify that there is a valid kerberos ticket associated with each packet. 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 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 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 have a matching IP. A straightforward implementation increased CPU usage to 97%, and reduced the throughput of the program by 73%, as pysmb became CPU-bound and not IO-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 bad practice for a program to request tickets from the Kerberos server long before the currently valid ticket expires. Thus, a useful invariant to check is that a successful authentication is not repeated until it expires or times out. A timeout occurs when there has been no activity in the last 300 seconds. To verify this invariant, we need to keep track of valid kerberos tickets, of kerberos requests, and of SMB activity.

The query we use is 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(KerberosTickets) ,
         k_new in extent(KerberosTickets) :
         k_old.valid and k_new.valid and
         k_old.timestamp<k_new.timestamp-10 and
         k_old.ip==k_new.ip and
         k_new.timestamp-max([p.time
            for p in $sent_packets
            if p.target_ip==k_new.ip])  < 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 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 a map of the latest packet sent to each IP address. We can do this because packets sent later are always later (time, and by extension, p.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, and not CPU bound. Just like before, to make it CPU bound requires checking invariants in a non-incremental manner. This results in a 96.9% CPU load (indicating that the program is CPU bound), and a corresponding increase in running time from 1302 to 8750 seconds.

The pysmb examples show that instrumenting complex programs in ways not assumed 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 Protocol implementation

BitTorrent (http://download.bittorrent.com/dl/) is a protocol for distributing files. Its advantage over plain HTTP is that when multiple downloads of the same file happen concurrently, the downloaders upload 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. It splits torrents into chunks, downloads all chunks from (likely different) peers, and then reassembles the original file(s) from chunks. Implementing a relatively complex protocol like BitTorrent may be error-prone, so we use our method to instrument an implementation and check it for potential errors.

No duplicate data. While it does not necessarily imply an error if one receives the same piece of data from two sources, doing so too often may mean that the client is using bandwidth inefficiently. To check for this, we use a rule that detects when the same data is received from two or more distinct sources, and logs the event without stopping the program. The log could then be later analyzed to determine if the duplicate data indicated a larger bug.

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

The rule makes sure that we get notified that we receive the same payload from two different IP addresses. It adds a queue into which all incoming packets get added upon construction, and then queries over this queue.

Experiments involved receiving a 10GB file from 30 peers, over a 100Mbit connection. We measured CPU load to determine the impact of the debugging rule. Without the rule, the average CPU load was 28.3%. With the rule applied, the CPU load increased to 36.1%. The small increase is due to the high selectivity of the `p1.payload==p2.payload` condition. Internally, the join is a combination of a reverse map lookup and hash join. Just like with pysmb, neither the instrumented, nor the uninstrumented versions were CPU-bound, both remained IO-bound. Allowing arbitrary code in the body of the `foreach` loop allows us to very quickly write a rule that just logs undesirable behaviour without stopping the program, as we have done before.

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 of packets, stored on the server. We write a query that detects when packets of the same chunk have a different payload (We compare the MD5 hashes of the objects).

BitTorrent uses instances of Packets (defined below) to exchange data, and, as we do not want the actual payload, but its MD5, we set the body field of the packets to None.

```
class Packet:
   def __init__ (self):
      self.md5=None
      self.source=None
      self.target=None
      self.chunk=None
```

	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)

Table 1. CPU utilization and wall time taken for experiments under differing optimizations.

```
      self.sent=False
      self.received=False
      self.body=None
```

The actual server is written as a single class with a rec_set field that maintains all packets sent and received by BitTorrent peers. We omit code that actually sets up listening on UDP port 636 (the port we chose), etc, as that code is very straightforward. The following query actually verifies that the invariant is not violated:

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

Finally, we present two InvTS rules that 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 rule as for handling `recieve`, with `recieve` 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 port 636
   $checking_socket=socket.open_udp(192.168.17.46,636)
   in global in function(myreceive(socket,packet):
      global $checking_socket
      $body=packet.body
      $arg.body=None
      $checking_socket.send(packet)
      packet.body=$body
do instead:
   myreceive($x, $p)
```

After applying the query and rules to the BitTorrent client and our server, we then 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, if running, run on another system) was 11%, vs 10% for the untransformed client. The server, when ran on the test machine (with the 5 clients run on a different machine) utilized 3.3% of the CPU with the instrumentation enabled, versus 2.7% with no instrumentation. This is a 13% penalty for verifying the invariant.

On a reliable connection we found no problems. On a connection that was bad (where we manually randomly injected changes into the packets sent by the peers) we found the errors before the BitTorrent verification algorithm, which requires bigger chunks, would find them.

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

5. Related work

There are two different areas this paper touches: runtime invariant verification [7], and incremental query result maintenance.

There is a large body of systems whose purpose is to verify temporal properties of subject programs. These include JavaMaC [12, 13], JPAX [11] , JNuke [1], and EAGLE [4]. These systems are very different from our system in that the invariants that they verify are written as some subset of LTL. Our system does not support writing invariants in terms of 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 does incur a performance penalty greater then dedicated systems designed to test LTL-based invariants, it is not a very significant performance penalty.

The category into which our system fits best is tools that use a side-effect free subset of their host language, extended with various operators such as quantifiers or set operations, to specify invariants to verify. The invariant specification languages include JML [17], Spec# [3], and Jahob [15]. These languages are specification languages. They are used to describe the invariants to be verified, and they rely on other tools to actually do the verification. For JML and Spec# there exist tools that allow the user to combine/compile the specification of the invariant and the subject program into a compiled program that, at runtime, verifies that the specified invariants hold. For Spec#, such a system is Boogie [2], for JML such systems include jmlc [6], jass [5], jmle [14], and DITTO [21]. Jahob has a run-time verifier in development [23].

Spec# does not support comprehensions[23]; nor does not support 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 [16]. 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 having support for liveness testing, as the trivial method of adding all created instances to a set does not take into account garbage collection.

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. This is in contrast with our system, which provides incremental maintenance of the value of the set comprehensions it supports. DITTO does provide incremental maintenance of some JML expressions, but it cannot incrementally maintain set comprehensions [21].

Another system, JQL [22], extends Java to support both comprehensions and extents, although it does not do this for the purposes of maintaining invariants, but rather for the purposes of querying over collections in Java. 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 query language, such as a set of all previously sent packets.

There has been a great amount of work done on incrementally maintaining invariants, e.g, [9, 19, 10, 18, 21, 20]. From these, especially relevant to this paper is the system we developed, (InvTS) [18], that applies rules that incrementally maintain query results. We use InvTS to apply rules that we derive from the user-provided queries that specify the invariants the user wishes to verify. The advantage of InvTS is its utilization of static analysis to reduce the runtime overhead of incrementally maintaining the results of queries. The rules we apply are automatically derived in a manner inspired by [20], which, while not being the only way to derive rules that perform incremental maintenance, turned out to be a very good fit to the queries whose results we wanted to maintain.

References

[1] C. Artho, V. Schuppan, A. Biere, P. Eugster, M. Baur, and B. Zweimüller. JNuke: Efficient Dynamic Analysis for Java. *Proc. CAV*, 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. *Proceedings of the Fourth International Symposium on Formal Methods for Components and Objects (FMCO 2005)*, 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. *Verification, Model Checking, and Abstract Interpretation: 5th International Conference, VMCAI 2004, Venice, Italy, January 11-13, 2004: Proceedings*, 2004.

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

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

[7] 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.

[8] D. Goyal. Transformational Derivation of an Improved Alias Analysis Algorithm. *Higher-Order and Symbolic Computation*, 18(1):15–49, 2005.

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

[10] A. Gupta, I. Mumick, and V. Subrahmanian. Maintaining views incrementally. *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 157–166, 1993.

[11] K. Havelund and G. Roşu. An Overview of the Runtime Verification Tool Java PathExplorer. *Formal Methods in System Design*, 24(2):189–215, 2004.

[12] M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a Run-time Assurance Tool for Java. *Proceedings of Runtime Verification (RV01)*, 55.

[13] 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.

[14] B. Krause and T. Wahls. jmle: A Tool for Executing JML Specifications Via Constraint Programming. *Lecture Notes in Computer Science*, 4346:293, 2007.

[15] V. Kuncak and M. Rinard. An overview of the Jahob analysis system: project goals and current status. *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, page 8, 2006.

[16] 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.

[17] G. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. Cok. How the Design of JML Accommodates Both Runtime Assertion Checking and Formal Verification. *Formal Methods for Components and Objects: First International Symposium, FMCO 2002, Leiden, The Netherlands, November 5-8, 2002: Revised Lectures*, 2003.

[18] Y. Liu, S. Stoller, M. Gorbovitski, T. Rothamel, and Y. Liu. Incrementalization across object abstraction. *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 473–486, 2005.

[19] R. Paige and S. Koenig. Finite Differencing of Computable Expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, 1982.

[20] T. Rothamel and Y. A. Liu. Automatic incrementalization of queries in object-oriented programs. `http://www.rothamel.us/osqTR.pdf`.

[21] A. Shankar and R. Bodík. DITTO: automatic incrementalization of data structure invariant checks (in Java). *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 310–319, 2007.

[22] D. Willis, D. Pearce, and J. Noble. Efficient Object Querying for Java. *Proc. of the European Conference on Object-Oriented Programming (ECOOP), Nantes, France*, 2006.

[23] K. Zee, V. Kuncak, M. Taylor, and M. Rinard. Runtime checking for program verification systems. *Workshop on Workshop on Runtime Verification (collocated with AOSD)*, 2007.