Practical Object-Oriented Back-in-Time Debugging*

Adrian Lienhard, Tudor Gîrba and Oscar Nierstrasz

Software Composition Group, University of Bern, Switzerland
{lienhard, girba, oscar}@iam.unibe.ch

Abstract. Back-in-time debuggers are extremely useful tools for identifying the causes of bugs. Unfortunately the "omniscient" approaches that try to remember *all* previous states are impractical because they consume too much space or they are far too slow. Several approaches rely on heuristics to limit these penalties, but they ultimately end up throwing out too much relevant information. In this paper we propose a practical approach that attempts to keep track of only the relevant data. In contrast to other approaches, we keep object history information together with the regular objects in the application memory. Although seemingly counter-intuitive, this approach has the effect that data not reachable from current application objects (and hence, no longer relevant) is garbage collected. We describe the technical details of our approach, and we present benchmarks that demonstrate that memory consumption stays within practical bounds. Furthermore, the performance penalty is significantly less than with other approaches.

1 Introduction

When debugging object-oriented systems, the hardest task is to find the actual root cause of the failure as this can be far from where the bug actually manifests itself [1]. In a recent study, Liblit *et al.* examined bug symptoms for various programs and found that in 50% of the cases the execution stack contains essentially no information about the bug's cause [2].

Classical debuggers are not always up to the task, since they only provide access to information that is still in the run-time stack. In particular, the information needed to track down these difficult bugs includes (1) how an object reference got here, and (2) the previous values of an object's fields. For this reason it is helpful to have previous object states and object reference flow information at hand during debugging. Techniques and tools like back-in-time debuggers, which allow one to inspect previous program states and step backwards in the control flow, have gained increasing attention recently [3,4,5,6].

The ideal support for a back-in-time debugger is provided by an *omniscient* implementation that remembers the complete object history, but such solutions are impractical because they generate enormous amounts of information. Storing the data to disk instead of keeping it in memory can alleviate the problem, but it only postpones the end, and it has the drawback of further increasing the runtime overhead. Current implementations such as ODB [3], TOD [4] or Unstuck [5] can incur a slowdown of factor 100 or more for non-trivial programs.

* Recipient of the ECOOP 2008 distinguished paper award. LNCS 5142, pp. 592-615.

The common strategy for discarding data is to delete the oldest data first, which inevitably leads to the problem that bugs that have their cause located far enough from their effect cannot be tracked down anymore [3]. Another strategy to address the memory problem is to generate less data by only instrumenting parts of the application [4]. In this case, however, the programmer must know upfront where the potential source of the problem is. This approach produces less data, but it still presents the problem that the data grows over time making it necessary to discard old data at some point.

In this paper, we attempt to answer the question: *How can we make back-in-time debugging practical, while still preserving all relevant information?* Our approach is to track historical information at the level of the virtual machine, and to keep the tracked information (*i.e.*, the object flows) in the same memory space as the regular application objects. A direct consequence of this approach is that information no longer reachable from the objects of the running application will be automatically garbage collected.

We extend the object memory model of conventional object-oriented virtual machines (such as Java or Smalltalk VMs) by representing object references as real objects on the heap. In this way we seamlessly integrate historical execution data into the object model of the virtual machine. The created object references and their mutual relationships capture side effects and the flow of objects through the system. This design provides the following benefits:

- The relevance of a datapoint is determined by the reachability of an object in the memory graph. Which history and how much of it is retained depends on the interconnectivity of the objects that capture historical execution data.
- Garbage collection of historical data comes "for free" since we can employ the usual garbage collector without any modifications to incrementally and efficiently delete no longer reachable data.

As our evaluation shows (Section 4), how much memory is consumed with our approach largely depends on the characteristics of the application. In some cases the data recorded does not grow indefinitely and hence in these cases recording can be turned on all the time. However, our approach does *not* guarantee that the virtual machine will never run out of memory — it only makes it less likely. In case the recorded data continues to accumulate over time, we run out of memory much later than with conventional approaches. In the latter case, we provide means to configure the recording to capture and remember less data, which can lead to a dramatic decrease in memory consumption.

To make back-in-time debugging truly practical, it is important not only to manage memory consumption, but also to keep the runtime overhead within reasonable limits. A slowdown of 100 can make a program unusable even for debugging. Unlike many other back-in-time debuggers, which rely on bytecode manipulation techniques and application-level logging, our implementation is at the virtual machine level and because of that the performance is significantly improved. From our experiments the worst case scenario led to a slowdown of only a factor of 7 compared to the original virtual machine.

The contributions of this paper are:

- An object model for object-oriented virtual machines with an explicit notion of references to capture and introspect historical execution data for back-in-time debuggers.
- An approach to capture past object state and object flow and to incrementally discard no longer relevant information by employing the garbage collector.
- Benchmarks for a modified Smalltalk virtual machine implementing our approach and an evaluation of the execution overhead and memory characteristics for real world applications.

Outline. In the following section we describe our approach to incorporate historical information into a high-level language virtual machine. In Section 3 we discuss our implementation, in Section 4 we present our evaluation and in Section 5 we discuss the results. We present the related work in Section 6 and conclude in Section 7.

2 Approach

Most back-in-time debuggers are based on tracing events emitted at the application level. This technique is commonly based on transforming bytecode to introduce sensors that emit events. We take a radically different approach by modifying the virtual machine to add the program's execution history to the object model. The model used to store the execution history is based on our previous works on analyzing dynamic data flows in the context of program comprehension [7,8,9].

2.1 The Basis: Representing Object References as Objects

The memory layout of objects in object-oriented virtual machines typically consists of a header for the class pointer, hash bits, GC flags, size etc. and a fixed number of fields containing object references and primitive values. In many virtual machines, object references are implemented as direct pointers, that is, an object reference is just the address of that object in memory. Examples are the Sun HotSpot VM, Jikes RVM (formerly known as the Jalapeño VM [10]), and the Squeak Smalltalk VM [11].

In the proposed object model we add a level of indirection by representing object references by so-called *alias* objects. We chose the term *alias* to discern it from an *object reference*, the concept it represents.

Figure 1.a illustrates the typical approach where an object reference is represented by a pointer, and Figure 1.b shows how in our object memory model the object reference is substituted by an alias object. Thus, the pointer stored in field_1 points to the alias and the alias has a pointer to the actual object. Aliases cannot be nested, that is, the object reference of an alias is always a direct pointer to a non-alias object. Aliases cannot only substitute reference values but also the undefined value and primitive values. In this case field_1 contains the primitive value (*e.g.*, tagged pointer for small integers).

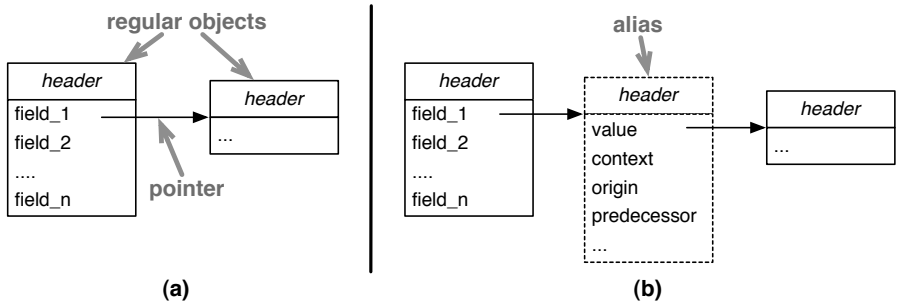

Fig. 1. (a) Typical object format with references as direct pointers and (b) proposed extension with references being optionally represented by alias objects.

Aliases have the following key properties that distinguish them from common objects:

- **Transparency.** Aliases are completely invisible at the application level. This means that the semantics of the language are not altered. For instance, method lookup, field access, or primitive operations are performed as if the actual object were referenced directly. To make the information of an alias accessible at the application level we use the concept of Mirrors [12].
- **Optionality.** The conventional direct pointer reference model (Figure 1.a) is still supported. This allows the recording of aliases to be switched on only when required (Section 2.2).
- **History.** Apart from the object pointer, an alias carries information about the object reference it represents. Through the relationship with other aliases, two main dimensions of object-oriented runtime behavior are captured: historical object state (Section 2.3) and object flow (Section 2.4).

Representing aliases directly as conventional objects allocated on the heap simplifies the internal object model of the virtual machine and allows us to use the standard garbage collector without needing to adapt it. For the same reason, many virtual machines represent classes and methods as internal objects.

2.2 Capturing Object References

In order to track object flow and reconstitute program states, we create an alias object whenever an object is:

1. allocated (referred to as *allocation alias*),
2. passed as parameter (*parameter alias*),
3. returned from a method invocation (*return alias*),
4. read from a field (*field read alias*),
5. read from an array (*array read alias*),
6. written into a field (*field write alias*), and
7. written into an array (*array write alias*).

The rationale is to capture all situations in which an object is made visible in a method invocation (1-5) or wherever a side effect is produced (6,7). Furthermore, we also capture the initialization of fields with the undefined value (referred to as *init alias*). By representing exceptions as objects, an allocation alias is created when an exception is thrown. Like this, also the propagation of exceptions is captured and can be traced.

Figure 2 illustrates the relationships among the entities Alias, MethodInvocation, and Value. With Value we refer to any type of value of the language (namely, objects, arrays, and primitive values, including the undefined value). A MethodInvocation represents a frame on the execution stack and also is a real object on the heap.

In contrast to other back-in-time debugging approaches, which typically collect and store data centrally as a trace of events, aliases are part of the object model. Like events, aliases capture historical execution data, but instead of ordering them in a temporal trace they are attached to object references. For example, in the case of writing to a field the alias objects are directly pointed to from the corresponding field of the object. For each value there can exist many aliases, whereas an alias always points to exactly one value.

Parameter aliases are referred to from the method invocation in which they replace the pointer to the actual parameter objects. The context of an alias is used to navigate to the method invocation in which the alias was created. To model the call stack, each method invocation holds onto its caller. In case of object allocation, return value, field read and array read, the aliases are used on the operand stack of the method invocation the same way as the objects they point to would be.

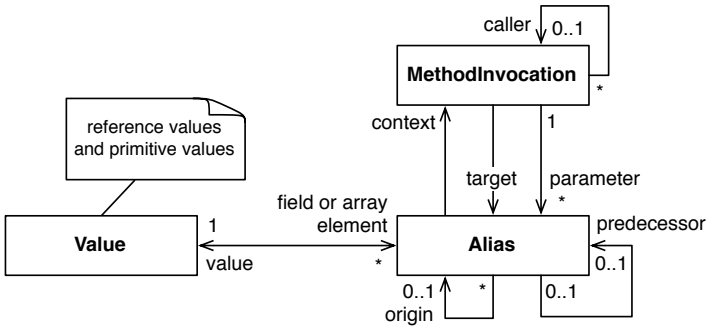

Fig. 2. Conceptual object model with aliases capturing historical execution data.

The predecessor of a field write alias is the field write alias of the value previously stored in the field (respectively the array write alias previously stored in the slot of the array). Only field and array write aliases have a predecessor.

The origin of an alias is the alias that was used to create the alias. For instance, the origin of a field read alias always is a field write alias because going back one step in the flow of an object from a field read alias always leads to the field write alias. Any alias

can be the origin of potentially many other aliases. Allocation aliases and init aliases are the only aliases without an origin.

In the following two sections we discuss the predecessor and origin relationships among aliases in more depth. Those two orthogonal relationships among aliases are key to our approach as they capture object state changes and as they track the flow of objects.

2.3 Remembering Historical Object State

An important historical dimension to capture is how the state of an object evolves. This allows a back-in-time debugger to answer the question: *What were the previous values of a field and where in the control flow were they assigned?* More precisely, we want to capture data that allows us to later determine which value was stored in a field of an object (or at a specific index of an array) at a particular point in time.

Figure 3 illustrates an example of a person object with the attribute name. When the object is allocated at the point in time t_1, the field is initially undefined. Later, at t_2, the string "Doe" is written into the field and at t_3 it is renamed to "Smith".

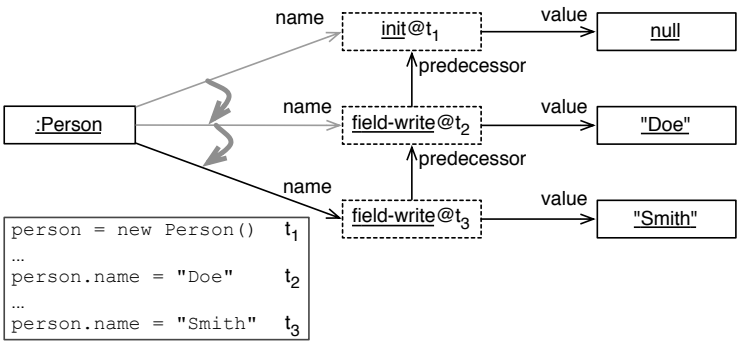

Fig. 3. Capturing historical object state through predecessor aliases.

In our model, the initial undefined value is captured by an alias of type *init* and all subsequent stores into the field are captured by aliases of type *field write* (or *array write* respectively). In the example the field first points to the alias of null, then to the alias of "Doe" and lastly to the alias of "Smith". The key idea is that each alias keeps a reference to its predecessor, that is, to the alias that was stored in the field beforehand. In this way, the alias pointed to from a field is the head of a linked list of aliases that constitute the history of that field.

Looking into the past. To go back in time, a selected process can be put into a state in which it "sees" the system as it happened to exist at a certain point in the past. Like this, accessing a historical value of a field is straightforward because when accessing a field (or array), the historical value is returned directly — just like the current value is normally returned.

Figure 4 shows pseudocode for the implementation of field access in the virtual machine. In case the current process has an activated back-in-time view, the predecessor list of the currently referenced alias is traversed backwards to the alias that was present in the field at the selected point in time.

```
if process.timestamp not defined then
  return x.f
else
  alias := x.f
  while alias.timestamp > process.timestamp and alias.predecessor is defined
    alias := alias.predecessor
  return alias
end if
```

Fig. 4. VM implementation of field access x.f with back-in-time capability.

In the example of Figure 3, accessing person.name at timestamp t_3 directly returns the alias of the string "Smith" whereas at t_1 an alias of the undefined value is returned.

With this model previous object state can be accessed very quickly (depending on the number of state changes of the field, which is typically a small number). Compared to other approaches, which need to reconstitute previous object state from a log or database, this is significantly faster (see Section 4).

2.4 Remembering the Flow of Objects

In addition to the historical object state dimension discussed above, we want to capture how objects propagate at runtime. The goal is to answer the second key question, which is: *How was this object passed here?* This means, for any object accessible in the debugger, we want to be able to inspect all origins up until the allocation of the object. This also allows us to find out where a particular value of a variable comes from. Furthermore, we also want to track the flow of the undefined value and any primitive values. Tracking the undefined value is important as null pointer exceptions can be hard to debug.

The way we track the flow of objects is similar to that of tracking past object states discussed above. In our model, all transfers of object references in the system are captured by the creation of aliases (when recording is turned on). Each alias in the system originates from an existing one, except for the allocation and init aliases, which are created when instantiating a class. Therefore, to capture how an object is passed through the system, each alias maintains a link to the alias from which it originates.

Figure 5 illustrates the flow of an account object in an execution trace (represented as a tree of method invocations where the callee points to the caller). A point represents an alias. An arrow from one alias to another shows the origin of the alias. Note that the actual flow of the object reference is opposite to the direction of the origin arrows.

Each alias is created in the context of the method invocation in which the object reference becomes visible. This means that for return values this is the calling method rather than the returning method. The parameter aliases are created at the callee site.

By means of the origin link of an alias we can track back how an object was passed to a method invocation in which a failure occurred. This helps one to understand how and why a possibly incorrect object reference has been propagated — even and especially if its flow spans the whole program execution and goes through fields and arrays.

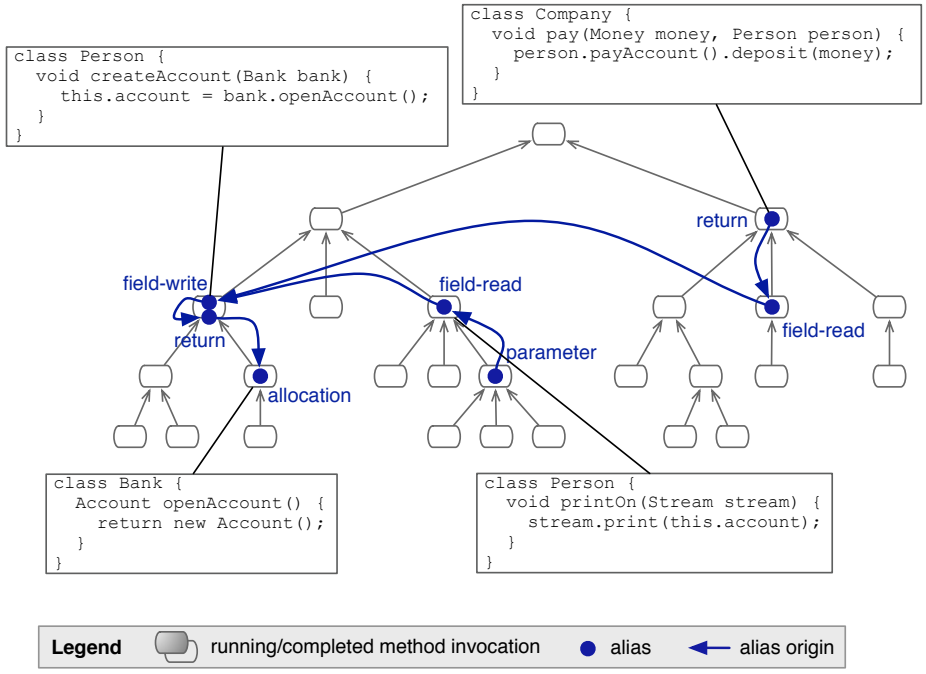

Fig. 5. Flow of an Account instance through an execution tree.

Introspecting object flows. Aliases are completely invisible at the application level because they forward all messages to the actual objects. Therefore, we have to provide other means to access object flow information than to send messages to an alias instance. We employ the concept of Mirrors [12] to introspect aliases. For each object reference that is represented by an alias instance, a mirror can be obtained through a primitive. A mirror is an object that provides an interface to access for example the origin and the predecessor of an alias, which in both cases returns a new mirror of the corresponding alias. In the same way we can access the method invocation context of an alias to get information about where in the control flow the alias was created. Our prototype implementation of the enhanced debugger uses this mechanism to navigate backwards in time.

2.5 The Effect of Garbage Collection

The upper part of Figure 6 illustrates the same execution trace and object flow illustrated by Figure 5. The currently active call stack is highlighted on the right side. The method invocations and the aliases are nodes in the memory graph, whereas the caller and origin arrows are the directed edges in this graph. The effect of a garbage collection is illustrated in the lower part of Figure 6.

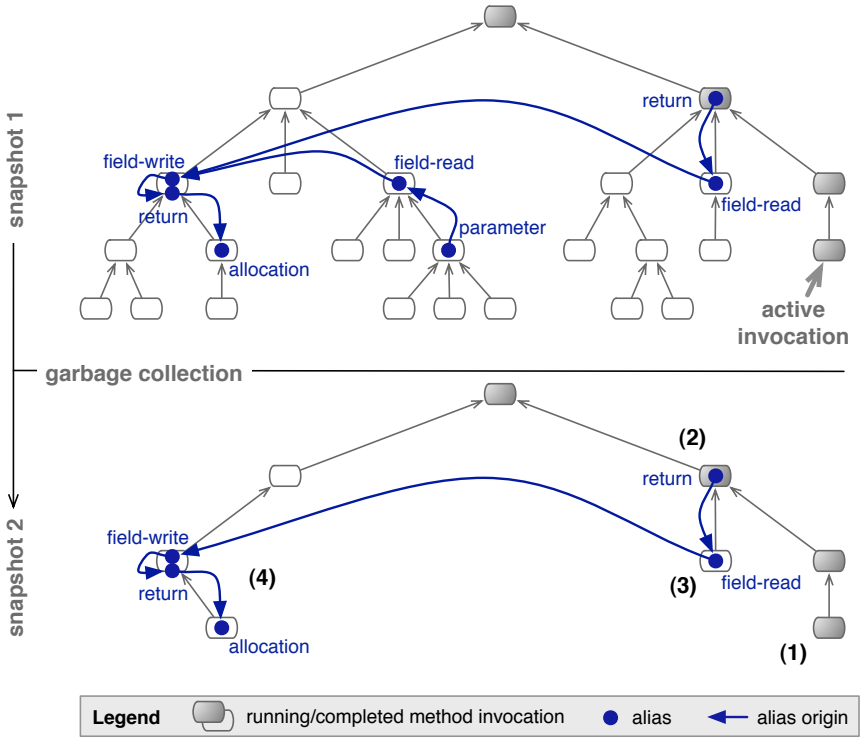

Fig. 6. Flow of an object through an execution tree and the effect of garbage collection.

In this example, the following objects survive the garbage collection:

- The current call stack is preserved since the active invocation (1) is considered a root object.
- The return alias (2) of the invocation payAccount() is not deleted since it is referred to from the operand stack of the active invocation pay().
- The invocation of payAccount() (3) is also preserved as it is the context of a field read alias and the field read alias is the origin of the return alias (2).

– The corresponding field read alias originates in a much earlier executed branch of the call tree where the account instance was written to a field after it was returned from the method invocation openAccount() (4).

The branch of the object flow (the field read and parameter aliases in the middle of the execution tree) does not survive the garbage collection. The reason is that no other object flow exists that would make a connection to the alias and invocation sub-graph. Also, many method invocations do not survive as they are not subject to relevant object flows and as they are not in the caller chain of a relevant invocation.

Figure 7 shows memory statistics from the execution of the Squeak bytecode compiler. In regular intervals we measured how many aliases have been allocated in total (solid line) and how many of those aliases still exist in memory (dashed line) over time. The effect of the garbage collection over the whole execution is a reduction of data by 70%. Both statistics in Figure 7 show the same run of the compiler, but with different garbage collector settings. On the left side, there are fewer GC cycles. For instance between 50ms and 120ms there is no GC activity and therefore both lines increase at the same rate. On the right side, between almost all sample steps there is a GC cycle.

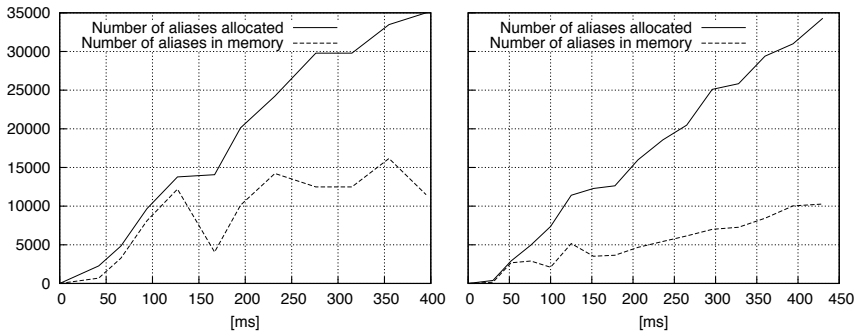

Fig. 7. Garbage collection discards 70% of the aliases in a run of the compiler. The right side shows the same execution but with a flatter curve due to more GC activity.

3 Implementation

We have extended the Smalltalk Squeak VM [11] with the recording capability and representation of the execution history in the object model as described in Section 2. The majority of the Squeak VM is implemented in a subset of Squeak Smalltalk, named Slang. The Slang source code is then translated to C to compile and link with the low-level, platform-specific C code. The Squeak VM implementation closely follows the specification given in the Smalltalk-80 Blue Book [13], except for the object memory format. Like most modern virtual machines, Squeak implements references as direct pointers rather than using an object table.

We implemented aliases as real objects of a new class Alias that has the fields value, context, origin and predecessor as illustrated in Figure 1 and Figure 2. In addition, Alias has an integer field that encodes information like the timestamp and the type of the alias (one of the 8 types described in Section 2.2).

Representing aliases as ordinary objects in memory has the advantage of simplifying the implementation. Most importantly, no changes to the object memory layout and to the garbage collector are necessary. The two main changes to the virtual machine are to allocate and initialize aliases, and to forward message sends to the actual target object in case the object is aliased. Aliases are created in the bytecode routines (*e.g.*, read and write aliases), on method invocation (parameter and return aliases) or when instantiating a class. There exist a few exceptional classes for which no aliases are created to simplify the implementation where aliasing is not important. Those classes are Process, Semaphore, MethodContext, BlockContext, and CompiledMethod.

Since method invocations are already represented as objects in Squeak (instances of the class MethodContext), to implement the model as illustrated in Figure 2, no further changes were required.

To optimize performance and space we implemented Alias as a compact class. That is, the object header of its instances consists of only a single 32-bit word and contains the index of its class in the compact classes table. This spares one word per alias instance, but more importantly, it allows the virtual machine to check whether an object is an alias or a real object by looking at the object header alone. The efficiency of this check is especially important because not every object reference is represented as an alias and hence this check has to be performed very frequently.

To generate the different kind of aliases when tracing is enabled, we extended various bytecode routines (*e.g.*, the store and fetch bytecodes), the method invocation behavior, and the class instantiation primitives. Small modifications had to be applied to many other primitives and bytecode routines that operate on the actual object rather than on the alias. In those cases the receiver and objects popped from the operand stack need to be unwrapped, for instance in arithmetic operations or the jump bytecode routines.

Overall, about 200 methods of the Slang implementation were modified or created. In comparison, the core of the virtual machine is implemented with about 750 methods (not counting platform specific code directly written in C and plugin code). Half of our changes were necessary due to the need of unwrapping aliases. Other parts of the virtual machine, for instance the memory format, method lookup, and the garbage collector, are not modified.

At the application level only very few extensions in system classes are required to support recording, to allow the user to control recording, and to introspect the execution history. First, the class Alias has to be loaded. It implements no methods and cannot be instantiated by the user. Second, the class Process is extended to allow the user to control recording at runtime. We added a field to Process which specifies the recording mode as well as the timestamp of its back-in-time point of view. At runtime, the behavior of the virtual machine then depends on these settings of the active process.

In addition, we implemented the class AliasMirror, which can be loaded to introspect the execution history. Mirrors are used by the graphical debugger, which we modified to be capable of moving backwards in time and to navigate backwards in the flow of

objects. The debugger accesses information about the flow or history of an object by requesting a mirror for the reference through which the object is made visible in the selected method invocation. There is no need to traverse the heap or perform a lookup in a trace to get the flow or history of an object, since this information is available through direct object references. A mirror on an object reference is created by the virtual machine through a primitive call. Using a mirror on an alias, the fields of the alias can be accessed. This behavior is implemented with a set of primitives in the virtual machine because any direct access to the alias would be performed on the actual object rather than on the alias instance.

4 Evaluation

In this section we evaluate our implementation from the point of view of the execution overhead (Section 4.1) and of the memory usage (Section 4.2). All experiments were performed on a MacBook Pro, 2.4GHz Intel Core 2 Duo, 4GB RAM, with Mac OS X 10.5.2.

4.1 Execution Overhead

Setup. To evaluate the execution overhead, we compare the performance of the modified Squeak virtual machine to the original virtual machine by means of several standard benchmarks. As a reference, we first executed the benchmarks in an original Squeak virtual machine (version 3.9-10), which we compiled using gcc 4.0.1. Then, we executed the same benchmarks using our modified virtual machine, which had been compiled under identical conditions. First, we took the benchmarks with the recording of historical data being turned off, and second with recording turned on. For each of the three cases the five benchmarks were executed 30 times, and before each execution we forced a full heap garbage collect.

Overview. The results of this comparison are shown in Table 1. The first three columns show the results of the benchmarks executed on our modified virtual machine without historical data recording, that is, no aliases are created. The remaining columns to the right show the results obtained from running the benchmarks with recording turned on. These overheads include the time that is needed to allocate and initialize alias instances, the additional time to forward message sends from aliases to normal objects and the additional time spent in the garbage collector.

The most important numbers are shown in the two Δ columns, which indicate the execution overhead of the benchmarks compared to the reference run of the unmodified standard virtual machine. The column *time* is the runtime of the benchmark and *%GC* indicates how much of this time is consumed by the garbage collector. The last two columns of the table show how many alias objects respectively method invocation objects are created (k indicates that both figures are given in 1000 objects). The figures in Table 1 are computed using the arithmetic mean of the 30 runs of each benchmark.

	Recording OFF			Recording ON				
Benchmark	Δ	time[s]	%GC	Δ	time[s]	%GC	aliases	methods
Tiny benchmark (bytecodes)	**1.02**	1.03	0.0	**2.26**	2.29	16.1	13773 k	8 k
Tiny benchmark (sends)	**1.20**	1.39	0.0	**2.06**	2.39	7.1	7881 k	11406 k
STones80 (low-level)	**1.12**	0.51	5.6	**1.53**	0.70	8.4	21600 k	4960 k
STones80 (medium-level)	**1.27**	0.38	0.4	**6.43**	1.91	46.0	59478 k	17077 k
Squeak macro benchmark	**1.16**	0.38	2.0	**6.91**	2.23	60.7	5532 k	669 k
Average	**1.15**	0.74	1.6	**3.84**	1.91	27.6	21653 k	6824 k

Table 1. In comparison with the original VM, the execution overhead of the modified VM averaged 15% when recording is disabled and the average slowdown when recording is turned on is 3.84 (see column Δ).

Benchmarks. We used five different benchmarks[1]. The first two rows show the results of two *Tiny benchmarks*, which measure how many bytecodes and message sends can be executed per second. The bytecode benchmark is based on a bytecode-heavy implementation of the "Sieve of Eratosthenes" whereas the message send benchmark is based on a send-heavy recursive calculation of Fibonacci numbers. The second and third rows show the results of the *STones80 benchmarks*, which are available for many different Smalltalk dialects. Whereas the low-level version mainly involves arithmetic operations, array operations, and object allocation, the medium-level version also performs recursive calls, collection and stream operations. The last row of Table 1 shows the results of the *Squeak macro benchmark*, which measures decompiling and then recompiling methods.

Discussion. The results of the benchmarks taken with disabled recording averaged 15%. These numbers are a good indication of the performance penalty caused by our virtual machine modifications. When recording is turned off, no aliases are allocated and hence no message sends have to be forwarded and no additional time is spent in the garbage collector. Interestingly, the overhead of the Tiny bytecodes benchmark is very low with an overhead of only 2%, while the overheads of the other benchmarks average between 12% and 27%. To find out whether this difference is significant or whether our modifications have no measurable influence on the performance of the bytecode benchmark, we performed the following statistical analysis.

We formulate the null hypothesis H_0 that the average runtime of the Tiny bytecode benchmark is not slower when executed on our modified VM (M) compared to the original VM (O), formally: $\mu_O \geq \mu_M$. The alternative hypothesis H_1 postulates that the average runtime of the benchmark is slower when being executed on our modified VM compared to the original VM, formally: $\mu_O < \mu_M$.

To test the hypotheses we apply the independent one-sided two-sample t-test [14] with an α value of 1% and 58 degrees of freedom. The variance requirement is fulfilled and both data sets are normally distributed (verified with the Kolmogorov-Smirnov

[1] The Tiny benchmarks can be found in the standard Squeak distribution. The STones80 and the Squeak macro benchmarks can be found in the Benchmarks package on http://map.squeak.org/

test). We calculated a t value of -16, which means that we can clearly reject the null hypothesis H_0 and accept the alternative hypothesis H_1 (the t distribution tells us that the probability that $t \leq -2.4$ is 1%). Therefore, we can conclude that the 2% slowdown of this benchmark is due to our modifications of the VM. Using the same method, we can draw the analogous conclusion for all other benchmarks. This result is not surprising as those runtimes are clearly distinct from the reference runtimes.

In the case of recording switched on, particularly noticeable are the big differences between the overheads of the first three low-level benchmarks compared to the other higher-level benchmarks. The overheads of the medium-level STones80 benchmark (factor 6.43) and the Squeak macro benchmark (factor 6.91) are more than three times as big as the overheads of the low-level benchmarks. One reason for this is that those benchmarks spend a significant percentage of their runtime in the garbage collector (46.0% and 60.7%). The high pressure on the garbage collector cannot be explained entirely by the higher rate by which alias and method invocation objects are created (31 million aliases/s for medium-level STones80 and 2.5 million for the macro benchmark). For instance, the low-level STones80 benchmark produces 30.7 million aliases/s but incurs a relatively low overhead. Rather, it is likely that the high pressure is due to the different memory usage characteristics, *e.g.*, how fast aliases can be garbage collected.

Summary. These benchmarks suggest that a significant overhead incurs because of the additional pressure on the garbage collector, which depends on the characteristics of memory usage. (Memory usage characteristics are further discussed in Section 4.2.) In turn, instantiating and initializing aliases seems to contribute not as much as the garbage collector to the overhead.

Without much optimization effort the overhead of our implementation when recording is switched off is just 15%. This suggests that with more aggressive performance optimizations a virtual machine enhancement for capturing execution history could potentially be incorporated into a standard distribution. This would allow users to switch recording on and off as required without needing to recompile code and restart the application with a different virtual machine.

4.2 Memory Usage

To further evaluate the practicality of our approach, we also investigated the characteristics of larger applications with respect to the amount of memory consumed. Of particular interest is whether the retained historical data increases steadily over time, or whether the amount of data is bounded by an upper value.

We expected this characteristic to be dependent on the type of application. For example, in applications with persistent objects that undergo many state changes, this is obviously not possible as all previous object states are retained until the objects themselves are garbage collected. In contrast, in applications where objects are used only temporarily it is possible that long running programs can be recorded without running out of memory.

To study different types of memory usages we chose the following three programs:

- *A program that allocates a large number of temporary objects that get garbage collected after some time.* We selected the Squeak bytecode compiler which we ran on 1000 classes. We expected that the history of objects generated to represent tokens, AST nodes and intermediate representation objects can be garbage collected after the bytecode of a class has been successfully emitted.
- *A program with a stable number of objects that undergo a large number of state changes.* We selected a gas tank simulator shipped as a Squeak demo. Each molecule in the tank is represented by an object and on each GUI update the position of the molecules, their velocities and directions are recalculated and changed. Since all previous positions of the molecules are remembered, we expected a lower effect of the garbage collector in comparison to the bytecode compiler.
- *A program with an existing object graph that is heavily accessed and modified.* We chose a commercial web content management system (CMS). The history of modifications of the object model and the behavior leading to it cannot be discarded after some time because the object model of the CMS is completely kept in memory.

Bytecode Compiler. Figure 8 shows virtual machine statistics taken from sampling the execution of the Squeak bytecode compiler. We compiled 1000 classes from the Squeak Smalltalk system which took 652 seconds when recording was turned on (compared to 168s when recording was disabled). In total the execution produced more than 2 billion aliases (solid line in Figure 8) and 443 million method invocations (not shown). On average, 3 million aliases are created per second. The actual number of aliases in memory was relatively low at an average of 2.9 million (notice the different scales of the left and right Y-axes). The maximum amount of memory allocated by the virtual machine was 317MB.

The temporal development of the number of regular objects (excluding alias objects) is similar to one of the number of aliases. The reoccurring pattern of growth and decline is caused by incremental and full garbage collect cycles.

The analysis of this application showed the expected behavior. The historical data kept in memory does not grow without limit because the compiling history of a class is discarded after the bytecode has been generated and emitted.

Gas tank simulation. Figure 9 shows the analysis of the following usage scenario of the gas tank simulation. First we started one instance of the simulator and paused it after the sample step #5. Then we started a second simulator with twice the number of molecules. The higher rate of aliases allocated from this time on is reflected in Figure 9. At step #10 we quit the second simulator and resumed the first one.

Quitting the second simulator has a big effect on the number of aliases retained in memory (see decline between steps #10 and #11). Since the objects of the second simulator are not accessible anymore, the remaining execution history is garbage collected. The same happens after quitting the simulator at the end of the analysis, where the number of aliases in memory drops to zero.

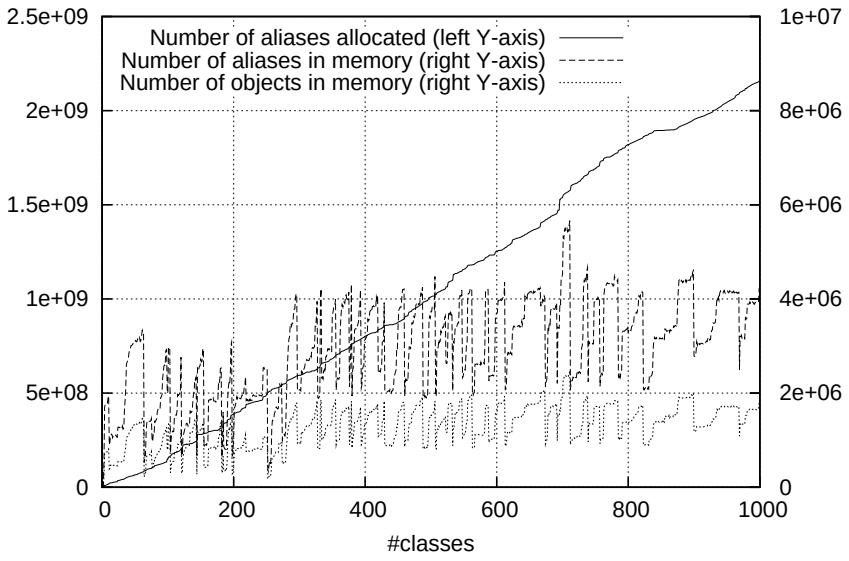

Fig. 8. Compiling 1000 classes (X-axis) produces more than 2 billion aliases, however, the number of aliases in memory stays below 6 million. Please note that because of the large differences between the number of allocated aliases (solid line) and the aliases and objects in memory (dashed and dotted lines), we use two scales: one for the solid line to the left and one for the dashed and dotted lines to the right.

A striking difference to the case of the bytecode compiler is that the number of aliases in memory grows with respect to the number of aliases allocated over time. As Figure 9 shows, the ratio is constantly 22% in the first half of the analysis up until the event where the second simulator instance was quit. This means, that for this application 78% of the aliases are garbage collected but the rest adds up in memory and eventually the virtual machine will run out of memory for long runs.

The execution history retained by our approach allows one to revert the state of objects that are currently accessible (or that are accessible through a past field reference of an accessible object). In case of the simulator this means that we can move back in time as long as the simulator user interface has not been closed. For instance we can set the point of view of its GUI process to a past point in time. This has the effect that the molecules are displayed where they were positioned at that time, and that also all settings of the simulator are reverted to their previous states. To find out how a position was calculated, we can follow back the flow of the corresponding point object to where it was allocated.

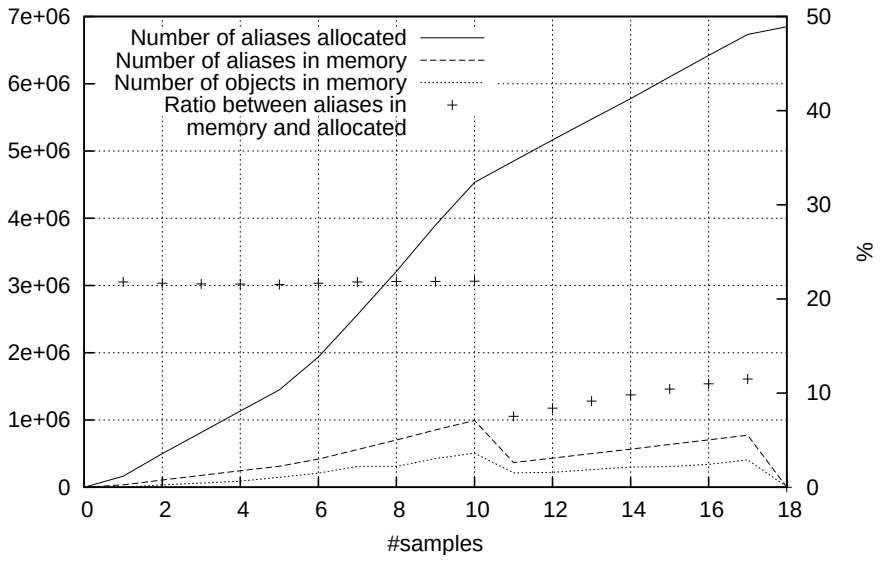

Fig. 9. Analysis of a gas tank simulator shows that 22% of the aliases allocated are retained in memory (19 samples with an interval of 3s each).

Content Management System. Figure 10 illustrates an analysis of a user session in Cmsbox[2], a commercial web content management system. The session consists of 26 user actions such as login, editing content, drag and drop, copy and paste elements, publish page etc.

We chose Cmsbox as a case study, because it stores all objects in memory rather than in a database, which makes it a worst case scenario for our approach. Indeed, as the figure shows, the aliases do increase steadily over time, the main reason being that more objects are added and retained in the model and in memory.

This experiment shows the limits of our approach. However, as described in Section 5.1 we can limit the effect of this phenomenon through selective recording of aliases.

5 Discussion

Memory consumption and performance can be further tuned by adjusting the level of detail of information gathered. We now look at several ways this can be done, and we discuss difficulties, limitations and potential optimizations of our implementation.

[2] http://www.cmsbox.com/

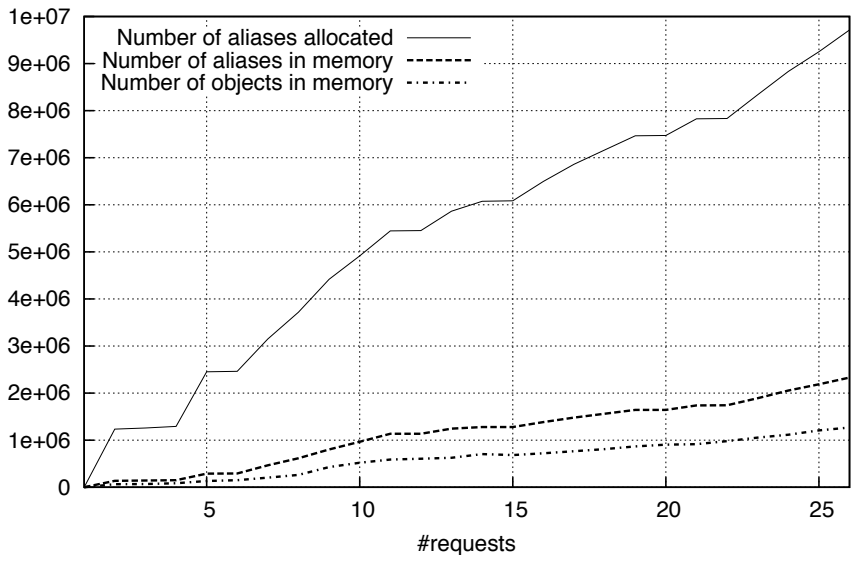

Fig. 10. Analysis of a user session in a Content Management System. After 26 requests, 24% of the allocated aliases are still in memory.

5.1 Capturing and Remembering Less Data

Depending on the usage of the back-in-time debugger, for instance in a testing or production environment, it can be necessary to further decrease memory consumption and lower the execution overhead. A common solution to reduce the amount of data recorded is to not instrument all code, for instance by excluding libraries and framework code. The effect is that in the code that is out of scope, no side effects are captured and the links of objects being passed through this code are lost.

We experimented with an alternative approach that is not based on structural scoping but on tuning how fast recorded data is discarded. In particular, in our implementation the user can change the behavior of the virtual machine by (a) disabling tracking of predecessor aliases, (b) disabling tacking of origin aliases, and (c) selecting which types of aliases are created. By default, all predecessor and origin aliases and all types of aliases are recorded.

For example by not tracking predecessors, we can reduce the consumed memory but still benefit from being able to inspect object flows. By means of such configurations we can provide the same functionality as the following two debugger extensions that have been proposed recently. They are specialized to a particular debugging task and hence only need to track a fraction of the whole execution history.

Reverse watchpoint, an approach proposed by Maruyama *et al.*, analyses the execution and moves the debugger to the last write access of a selected variable by re-executing the program from the beginning [6]. This technique automates the task of finding where a variable was erroneously written and then moves the debugger to that

point. With our approach, finding where a variable was written means to move one step back in the object flow from a field read alias to its origin, which is the field write alias. If we want to gather exactly this information, we can disable predecessors, and restrict the recording to create only field write and array write aliases. The effect is that for each field the most recent write alias is available with the execution stack in which it was created. When writing to a field, the previous write alias of the field or array can be discarded immediately because it is not referenced as a predecessor or origin alias.

Our benchmarks show that with this minimal configuration, we achieve a very low execution overhead that is only insignificantly higher than the base slowdown of 15%, which the virtual machine incurs when recording is turned off completely. In comparison, Maruyama report a slowdown of their technique of about 400% [6]. Figure 11 shows that with this reduced configuration only a fraction of the aliases are remembered, compared to the aliases remembered using the standard configuration as illustrated in Figure 10.

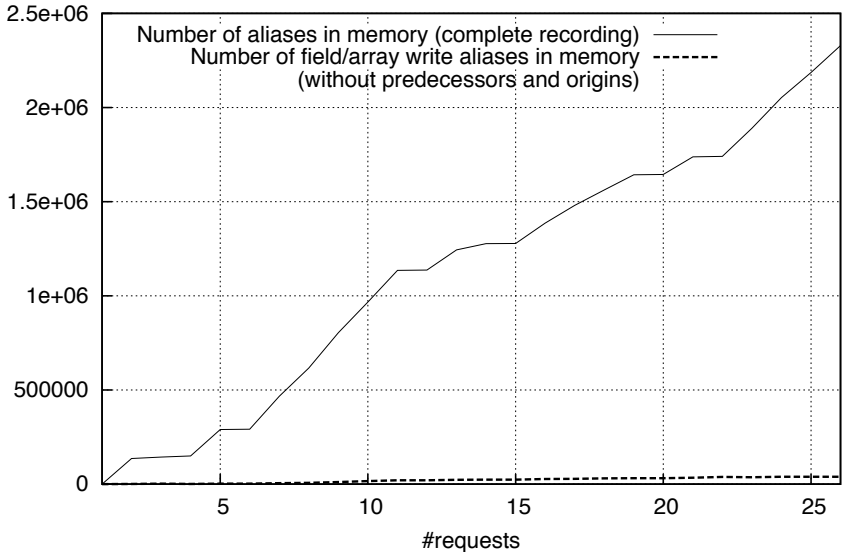

Fig. 11. Comparison of the number of aliases retained in memory with the default configuration compared to the configuration where only the last write alias of each field and array slot is remembered (same run of the CMS as in Figure 10).

Origin tracking of null values, proposed by Bond *et al.*, is a very efficient technique to track the method in which an undefined values originates to support debugging the well-known problem of null pointer exceptions [15]. We can do the same by only tracking aliases of type *undefined*, which again incurs a similar overhead to the configuration discussed above. In comparison, the approach of Bond *et al.* adds an overhead

of only 4%. The low overhead is made possible by only tracking undefined values, which allows for "value piggybacking", a technique to store origin information directly in pointers (in this case in null pointers).

5.2 Remembering Control Flow Dependencies

With the default configuration our approach retains information about the flow of objects and previous states of objects. In contrast, conventional back-in-time debuggers typically record and store the complete execution history (until they run out of memory). While our approach records the same data, it discards most of it within a very short time and only remembers what is relevant for the object flow and historical states of objects that are accessible at the current point of execution. To provide enough context, with each alias, the method invocation where it is used is retained, including the execution stack with all target objects and objects passed as parameters.

Still, depending on the kind of bug, it is possible that relevant information is missing. In particular, no links between aliases exist to represent the fact that the value of a variable influences the value of another variable. For example, in the statement "if x.f then y.f = 1" there is no dependency link between the field read alias x.f and the field write alias y.f. In case the value of y.f turns out to be incorrect because of the unexpected execution of a branch, it is possible that x.f has already been discarded. However, this does not happen if x is the target object of the method or one of its parameters. In this case it is referenced from the method invocation which in turn is referenced as the context of the field write alias stored in y.f. (How exactly aliases and method invocations refer to each other is illustrated in Figure 2.)

We decided not to explicitly capture such control flow dependencies because we believe that the most difficult bugs in object-oriented programs are caused by subtle inconsistencies in object graphs and by the propagation of unexpected object references. However, it would be possible to extend our approach to also include dependency relationships among aliases. Each alias would need to maintain a list of other aliases it depends on, similar to the predecessor and origin relationships. The dependence information could be computed by an intra-procedural static analysis. An inter-procedural analysis would not be necessary since this dependency is indirectly captured by the link of an alias to the target of the invocation in which it is created (alias → context → target).

5.3 Limitations and Potential Optimizations of the Implementation

Not freed memory. Using our back-in-time debugger we noticed a couple of times that parts of the history were unexpectedly not garbage collected. The reason turned out to be that the program execution in those cases produced subtle side effects on global state. The effect is that the part of the execution history that produced the side effect is not garbage collected as long as the global state that was modified exists. We observed three cases of this problem: singletons, caches and writing to a log console (strings passed from the application are stored in the console stream and hence retain links to the execution history where they originate). While in some cases this can be the desired behavior (in case of the cache or singleton), it may be undesired in other

cases (the console). To remedy the undesired cases we can simply disable recording of the appropriate methods or classes. The real difficulty, however, is to first find the cause of such a problem. What is missing are high-level views to inspect and navigate the recorded data.

Capturing non-word size data. A limitation of our implementation is that the history of values stored as non-word data are not captured. For instance, in a byte array where four bytes are stored per word, we cannot use the approach of exchanging a value with an alias indirection because the alias pointer requires 4 bytes. Typically this is not problematic since Squeak uses non-word fields in most cases to represent internal data of objects only, such as float objects, large integer objects, or strings.

Potential optimizations of our implementation. As our benchmarks show, the slowdown is mainly caused by the additional garbage collector activity. An optimization of the garbage collector to better cope with the special characteristics of our virtual machine would improve the performance but is not straightforward to realize.

A different optimization that would also improve the performance of the virtual machine when tracing is turned off is to use different sets of bytecode routines. The current implementation uses conditionals in bytecode routines and primitives to execute code depending on the tracing state of the current thread. Implementing two sets of bytecode routines would allow the virtual machine to switch jump tables when recording is toggled.

The memory consumption of our implementation could be slightly optimized by distinguishing between field/array write aliases and the other types of aliases. Field write and array write aliases require the predecessor field to hold onto historical state, whereas the other aliases do not need this field. Using two different classes of aliases would be simple to implement and would save one word per non-historical alias instance.

6 Related Work

Logging-based approaches. The most common approach to implementing back-in-time debuggers has been to create a trace log of the program execution. ZStep95 is a reversible debugger for Lisp that provides animated views but does not address performance and scalability issues [16]. Lewis proposed ODB [3], a back-in-time debugger for Java, and Hofer proposed Unstuck [5], a similar proof of concept implementation for Squeak Smalltalk. Both approaches have in common that they keep the log history in memory and hence can only record and store the complete history for a short period of time. ODB allows one to set a fixed limit on the number of events and it then discards older events when the limit is reached.

A more scalable approach has recently been proposed by Pothier *et al.* [4]. Their back-in-time debugger, TOD, addresses the space problem by storing execution events in a distributed database. While this approach has the benefit that no data is lost, its drawback is that it requires extensive hardware power, which is not available for many developers today. To cope with the data generated by a CPU-intensive program, 10 database nodes in a server cluster are required. Also, the approach has a performance overhead of a factor 113 in the worst case, which is approximately the same as the one of ODB for the same benchmark [4].

In comparison, the performance of our approach is about one order of magnitude better. On the one hand, this is because our approach is implemented at the virtual machine level, whereas all previously mentioned approaches are based on bytecode instrumentation. On the other hand, as our approach stores historical data directly in the application memory, it does not require any additional logging facility to gather and store data. As a side effect, our representation of historical information is also very space efficient. For example, there is no need to assign identifiers to objects or to serialize objects since they exist in memory and can be referred to directly by pointers.

Outside of research, back-in-time debuggers have unfortunately not been widely adopted yet. An example of a commercial back-in-time debugger is Omnicore's CodeGuide[3]. It is also based on bytecode instrumentation and its execution history, which is kept in memory, is limited to the few last thousand events. An interesting aspect of CodeGuide is that only methods containing breakpoints and methods close to them in the control flow are instrumented to keep the runtime overhead low.

Related to logging-based back-in-time debugging is *query-based debugging*. In those approaches the user formulates a query in a higher-level language that is then applied to the logged data [17,18,19,20]. Queries can test complex object interrelationships and sequences of related events. Approaches exist that execute the query at runtime, which can improve performance because no history has to be stored [21]. Our approach, like other back-in-time debugging approaches, does not support querying for complex relationships in the history, but in return it incurs a much smaller execution overhead.

Replay-based approaches. A different approach for implementing back-in-time debuggers is to replay the debugged program until a desired point in the past. To optimize the time required to reach a particular point in the past, many approaches take periodic state snapshots, for instance Bdb [22] and Igor [23]. The main advantage of replay-based approaches over logging-based approaches is their low performance overhead (roughly 2 times for Bdb and 4 times for Igor). The disadvantage of those kinds of approaches is that moving backwards in time can be very slow because the program has to be partly re-executed. This issue has been addressed in a recent publication by Xu *et al.* [24]. Our approach can access past object state almost instantly because it only needs to look up the appropriate alias in the predecessors chain (as described in Section 2.3). An open issue of replay-based approaches is that of deterministic replay, which cannot be guaranteed by all approaches if the program depends on external resources or if it is multithreaded.

The approach of taking (incremental) memory snapshots and replaying has also been used in the Leonardo virtual machine [25], a virtual machine based approach for assembly-like languages that features reversing program state. Similar to our approach, programs slow down by a factor of 6 in the worst case. However, the Leonardo virtual machine does not support inspecting object flows and it does not provide a strategy to discard data.

[3] http://www.omnicore.com/

7 Conclusions and Future Work

In this paper we tackle the problem of how to make back-in-time debugging practical by (1) keeping memory consumption within reasonable bounds by only keeping track of still-relevant past data, and (2) reducing the run-time overhead by implementing recording at the virtual machine level. Our approach does not store all data, but instead it focuses on remembering the history of the objects that are still referenced in the current program state. Our solution makes use of the garbage collector to release the objects that are not referenced anymore in the program and that are not relevant anymore in the program's history.

Benchmarks have shown significant improvements over existing approaches. First, the memory consumption is confined to an upper bound limit in the best case, or grows slowly in the worst case. However, for the worst case scenario, we can configure the recording to capture and remember less data, which can lead to a dramatic decrease in memory consumption (*e.g.*, 55 times fewer aliases when just remembering the last field write alias). Second, performance is in the worst case 7 times slower than a regular execution. Furthermore, the modified virtual machine with tracing switched off introduces only modest overhead (*e.g.*, in our benchmarks, it introduces an average of 15%) as compared with a regular one.

The results we obtained are very promising and we envision several paths to improve them. First, we want to provide more control for adjusting the level of detail depending on the static structure. For instance, we can gather more data in code that is young and hence is more likely to have defects. Second, we want to experiment with instrumentation mechanisms that can be adapted at runtime. For example, we can increase the recording detail when an error is detected for the followup runs, or we can decrease recording detail when memory gets low. Lastly, we want to further investigate the effectiveness of our approach, for instance to identify potential limitations of its usability due to missing control flow dependencies.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Analyzing, capturing and taming software change" (SNF Project No. 200020-113342, Oct. 2006 - Sept. 2008), and the financial support of ESUG. We would like to thank Stéphane Ducasse and David Röthlisberger for their help in reviewing drafts of this paper.

References

1. Zeller, A.: Why Programs Fail: A Guide to Systematic Debugging. Morgan Kaufmann (October 2005)
2. Liblit, B., Naik, M., Zheng, A.X., Aiken, A., Jordan, M.I.: Scalable statistical bug isolation. In: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI'05), New York, NY, USA, ACM (2005) 15–26
3. Lewis, B.: Debugging backwards in time. In: Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG'03). (October 2003)

4. Pothier, G., Tanter, E., Piquer, J.: Scalable omniscient debugging. Proceedings of the 22nd Annual SCM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'07) (2007) To appear, ACM.
5. Hofer, C., Denker, M., Ducasse, S.: Design and implementation of a backward-in-time debugger. In: Proceedings of NODE'06. Volume P-88 of Lecture Notes in Informatics., Gesellschaft für Informatik (GI) (September 2006) 17–32
6. Maruyama, K., Terada, M.: Debugging with reverse watchpoint. In: Proceedings of the Third International Conference on Quality Software (QSIC'03), Washington, DC, USA, IEEE Computer Society (2003) 116
7. Lienhard, A., Greevy, O., Nierstrasz, O.: Tracking objects to detect feature dependencies. In: Proceedings International Conference on Program Comprehension (ICPC'07), Washington, DC, USA, IEEE Computer Society (June 2007) 59–68
8. Lienhard, A., Ducasse, S., Gîrba, T.: Object flow analysis — taking an object-centric view on dynamic analysis. In: Proceedings of the 2007 International Conference on Dynamic Languages (ICDL'07), New York, NY, USA, ACM Digital Library (2007) 121–140
9. Lienhard, A., Gîrba, T., Greevy, O., Nierstrasz, O.: Test blueprints – exposing side effects in execution traces to support writing unit tests. In: 12th European Conference on Software Maintenance and Reengineering (CSMR'08), IEEE Computer Society Press (2008) 83–92
10. Alpern, B., Attanasio, C.R., Cocchi, A., Lieber, D., Smith, S., Ngo, T., Barton, J.J., Hummel, S.F., Sheperd, J.C., Mergen, M.: Implementing jalapeño in java. In: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'99), New York, NY, USA, ACM (1999) 314–324
11. Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., Kay, A.: Back to the future: The story of Squeak, a practical Smalltalk written in itself. In: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97), ACM Press (November 1997) 318–326
12. Bracha, G., Ungar, D.: Mirrors: design principles for meta-level facilities of object-oriented programming languages. In: Proceedings of OOPSLA '04, ACM SIGPLAN Notices, New York, NY, USA, ACM Press (2004) 331–344
13. Goldberg, A., Robson, D.: Smalltalk 80: the Language and its Implementation. Addison Wesley, Reading, Mass. (May 1983)
14. Kanji, G.K.: 100 Statistical Tests. SAGE Publications (1999)
15. Bond, M.D., Nethercote, N., Kent, S.W., Guyer, S.Z., McKinley, K.S.: Tracking bad apples: reporting the origin of null and undefined value errors. In: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications (OOPSLA'07), New York, NY, USA, ACM (2007) 405–422
16. Lieberman, H., Fry, C.: ZStep 95: A reversible, animated source code stepper. In Stasko, J., Domingue, J., Brown, M.H., Price, B.A., eds.: Software Visualization — Programming as a Multimedia Experience, Cambridge, MA-London, The MIT Press (1998) 277–292
17. Martin, M., Livshits, B., Lam, M.S.: Finding application errors and security flaws using pql: a program query language. In: Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05), New York, NY, USA, ACM Press (2005) 363–385
18. Lencevicius, R., Hölzle, U., Singh, A.K.: Query-based debugging of object-oriented programs. In: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming (OOPSLA'97), New York, NY, USA, ACM (1997) 304–317
19. Potanin, A., Noble, J., Biddle, R.: Snapshot query-based debugging. In: Proceedings of the 2004 Australian Software Engineering Conference (ASWEC'04), Washington, DC, USA, IEEE Computer Society (2004) 251

20. Ducasse, S., Gîrba, T., Wuyts, R.: Object-oriented legacy system trace-based logic testing. In: Proceedings of 10th European Conference on Software Maintenance and Reengineering (CSMR'06), IEEE Computer Society Press (2006) 35–44
21. Lencevicius, R., Hölzle, U., Singh, A.K.: Dynamic query-based debugging. In Guerraoui, R., ed.: Proceedings of European Conference on Object-Oriented Programming (ECOOP'99). Volume 1628 of LNCS., Lisbon, Portugal, Springer-Verlag (June 1999) 135–160
22. Feldman, S.I., Brown, C.B.: Igor: a system for program debugging via reversible execution. In: Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging (PADD'88), New York, NY, USA, ACM (1988) 112–123
23. Boothe, B.: Efficient algorithms for bidirectional debugging. In: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation (PLDI'00), New York, NY, USA, ACM (2000) 299–310
24. Xu, G., Rountev, A., Tang, Y., Qin, F.: Efficient checkpointing of java software using context-sensitive capture and replay. In: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC-FSE'07), New York, NY, USA, ACM (2007) 85–94
25. Demetrescu, C., Finocchi, I.: A portable virtual machine for program debugging and directing. In: Proceedings of the 2004 ACM symposium on Applied computing (SAC'04), New York, NY, USA, ACM (2004) 1524–1530