Elephant Tracks II: Practical, Extensible Memory Tracing

Xuanrui (Ray) Qi

Department of Computer Science,
Tufts University

Senior Honors Thesis Defense
May 3, 2018

Committee: Samuel Z. Guyer, Kathleen Fisher

Goals of my thesis

Make memory tracing great again!

1. Make memory tracing fast(er)
2. Make memory tracing extensible and eventually available for languages other than Java
3. Develop new, language-agnostic techniques to run the Merlin algorithm

What is memory tracing?

- **Complete** chronological record of what happened in the heap
- Procedure entry & exit, allocations, pointers updates/overwrites, deaths, etc.

Why do we need memory traces?

- In a world where all memory is manually managed...

Why do we need memory traces?

- In a world where all memory is manually managed...
- Do we know when memory is allocated?

Why do we need memory traces?

- In a world where all memory is manually managed...
- Do we know when memory is allocated? *Yes! We allocated that.*

Why do we need memory traces?

- In a world where all memory is manually managed...
- Do we know when memory is allocated? *Yes! We allocated that.*
- Do we know when objects die?

Why do we need memory traces?

- In a world where all memory is manually managed...
- Do we know when memory is allocated? *Yes! We allocated that.*
- Do we know when objects die? *Hopefully! If not, we will forget to deallocate them and create leaks.*

Why do we need memory traces?

- In a world where all memory is manually managed...
- Do we know when memory is allocated? *Yes! We allocated that.*
- Do we know when objects die? *Hopefully! If not, we will forget to deallocate them and create leaks.*
- What happens if we reference a deallocated object or an invalid pointer then?

Why do we need memory traces?

- In a world where all memory is manually managed...
- Do we know when memory is allocated? *Yes! We allocated that.*
- Do we know when objects die? *Hopefully! If not, we will forget to deallocate them and create leaks.*
- What happens if we reference a deallocated object or an invalid pointer then? *We don't know exactly. But likely our program will blow up.*

Why do we need memory traces?

- In a world where all memory is heap-allocated and automatically managed...

Why do we need memory traces?

- In a world where all memory is heap-allocated and automatically managed...
- What happens if we reference a deallocated object or an invalid pointer address?

Why do we need memory traces?

- In a world where all memory is heap-allocated and automatically managed...
- What happens if we reference a deallocated object or an invalid pointer address? *Nothing, because it can't happen. Yay!*

Why do we need memory traces?

- In a world where all memory is heap-allocated and automatically managed...
- What happens if we reference a deallocated object or an invalid pointer address? *Nothing, because it can't happen. Yay!*
- But do we know when memory is allocated?

Why do we need memory traces?

- In a world where all memory is heap-allocated and automatically managed...
- What happens if we reference a deallocated object or an invalid pointer address? *Nothing, because it can't happen. Yay!*
- But do we know when memory is allocated? *Sometimes...*

Why do we need memory traces?

- In a world where all memory is heap-allocated and automatically managed...
- What happens if we reference a deallocated object or an invalid pointer address? *Nothing, because it can't happen. Yay!*
- But do we know when memory is allocated? *Sometimes...*
- Do we know when objects die?

Why do we need memory traces?

- In a world where all memory is heap-allocated and automatically managed...
- What happens if we reference a deallocated object or an invalid pointer address? *Nothing, because it can't happen. Yay!*
- But do we know when memory is allocated? *Sometimes...*
- Do we know when objects die? *Almost never...*

Why do we need memory traces?

- In a world where all memory is heap-allocated and automatically managed...
- What happens if we reference a deallocated object or an invalid pointer address? *Nothing, because it can't happen. Yay!*
- But do we know when memory is allocated? *Sometimes...*
- Do we know when objects die? *Almost never...*
- Do we want to know these details, then?

Why do we need memory traces?

- In a world where all memory is heap-allocated and automatically managed...
- What happens if we reference a deallocated object or an invalid pointer address? *Nothing, because it can't happen. Yay!*
- But do we know when memory is allocated? *Sometimes...*
- Do we know when objects die? *Almost never...*
- Do we want to know these details, then? *Yes!*

What can we use memory traces for?

- Evaluating GC performance
- Help develop new GC algorithms
- Learn new facts about object lifetime patterns (Veroy and Guyer, 2017)
- Find memory leaks (Jensen et al., 2015)
- Help programmers understand their memory footprint

How to get memory traces?

- Through dynamic analysis/instrumentation

How to get memory traces?

- Through dynamic analysis/instrumentation
- Generating records on each allocation, pointer update, procedure entry/exit, etc.

How to get memory traces?

- Through dynamic analysis/instrumentation
- Generating records on each allocation, pointer update, procedure entry/exit, etc.
- What about deaths? They aren't obvious through dynamic analysis!

The solution

- Compute them!

The solution

- Compute them!
- Compute each object's **idealized death time** using the Merlin algorithm (Hertz et al., 2006, 2002).

The solution

- Compute them!
- Compute each object's **idealized death time** using the Merlin algorithm (Hertz et al., 2006, 2002).
- Idealized death time = latest time at which an object is shown possible to be reached.

Merlin

$$T_d(o) = \max(T_s(o), \{T_d(p) \mid \forall p : p \to o\})$$

$T_s(o)$: last-accessed timestamp of object
$T_d(o)$: "idealized death time" of object

Merlin

$$T_d(o) = \max(T_s(o), \{T_d(p) \mid \forall p : p \to o\})$$

$T_s(o)$: last-accessed timestamp of object
$T_d(o)$: "idealized death time" of object

The death time of an object is the max of:

- the last time it was accessed; and
- the death times of all objects pointing to it

Merlin: the algorithm

- Use an iterative method to compute death times
- "Propagate" timestamps by depth-first search
- Example on the board

How do we trace program execution exactly?

- **Instrument** the program

How do we trace program execution exactly?

- **Instrument** the program
- Insert code on-the-fly to generate runtime traces

How do we trace program execution exactly?

- **Instrument** the program
- Insert code on-the-fly to generate runtime traces
- Important: whenever an object is used, emit a "witness" record

How do we trace program execution exactly?

- **Instrument** the program
- Insert code on-the-fly to generate runtime traces
- Important: whenever an object is used, emit a “witness” record
- Analyze traces to generate death records

Can't we already do this?

- Yes, you're right. There's Elephant Tracks (Ricci et al., 2013).

Can't we already do this?

- Yes, you're right. There's Elephant Tracks (Ricci et al., 2013).
- But it's slow, heavyweight, and not very portable...

Can't we already do this?

- Yes, you're right. There's Elephant Tracks (Ricci et al., 2013).
- But it's slow, heavyweight, and not very portable...
- And it only works for Java

Can't we already do this?

- Yes, you're right. There's Elephant Tracks (Ricci et al., 2013).
- But it's slow, heavyweight, and not very portable...
- And it only works for Java
- That's why we need a new tool...

Elephant Tracks II architecture

- Frontend: trace program, generate execution records, timestamp events, etc.

Elephant Tracks II architecture

- Frontend: trace program, generate execution records, timestamp events, etc.
- Backend: do trace analysis, compute death records

Elephant Tracks II architecture

- Frontend: trace program, generate execution records, timestamp events, etc.
- Backend: do trace analysis, compute death records
- Pluggable architecture: one backend, multiple frontends
- Separates complex computations from slow tracing

Frontend implementation

- Currently, only Java frontend implemented
- Uses **JVMTI** to instrument program bytecode *at runtime*
- Uses **JNIF** (Mastrangelo and Hauswirth, 2014) to manipulate bytecode

Event detection in ET2/Java

- Event detection: finding events that need to be traced

Event detection in ET2/Java

- Event detection: finding events that need to be traced
- ET2/Java trace generation is *not completely dynamic*

Event detection in ET2/Java

- Event detection: finding events that need to be traced
- ET2/Java trace generation is *not completely dynamic*
- Bytecode search for certain key instructions

Instrumenting Java programs

```
52:  aload_1
53:  invokevirtual  #9
56:  goto           89
59:  aload_0
60:  getfield       #3
63:  ifnonnull      81
66:  aload_0
```

Instrumenting Java programs

```
52: aload_1
53: invokevirtual #9
##: invokestatic (method entry)
56: goto          89
59: aload_0
60: getfield      #3
##: invokestatic (witness)
63: ifnonnull     81
66: aload_0
```

Main optimizations

- **“Native” bytecode manipulation**
  ET: bytecode goes into a separate Java process, gets rewritten, and then sent back (very slow)
  ET2: bytecode gets rewritten directly in the JVMTI agent (much better)

Main optimizations

- **“Native” bytecode manipulation**
  ET: bytecode goes into a separate Java process, gets rewritten, and then sent back (very slow)
  ET2: bytecode gets rewritten directly in the JVMTI agent (much better)
- **Java-based instrumentation**
  ET: each instrumentation call is an FFI call to a C++ function (expensive)
  ET2: everything happens in Java (leverages JIT)

The backend

- Also called the **GC simulator**
- Workflow: “execute” the trace, run GC simulation as appropriate, compute death times after each GC

The backend

- Also called the **GC simulator**
- Workflow: "execute" the trace, run GC simulation as appropriate, compute death times after each GC
- Few assumptions about the memory model!

The backend

- Also called the **GC simulator**
- Workflow: “execute” the trace, run GC simulation as appropriate, compute death times after each GC
- Few assumptions about the memory model!
- Only assumptions: must have heap-allocated blocks, pointers, and GC

Simulating GC

- Problem: no exact knowledge of GC roots inside the traces, so can't run GC

Simulating GC

- Problem: no exact knowledge of GC roots inside the traces, so can't run GC
- Solution: use a approximate, conservative strategy!

Simulating GC

- Problem: no exact knowledge of GC roots inside the traces, so can't run GC
- Solution: use a approximate, conservative strategy!
- Treat everything possibly alive in the current context as GC roots

"Conservative root searching"

- Keep a stack that simulates the call stack

"Conservative root searching"

- Keep a stack that simulates the call stack
- Generate records for each parameter to procedure on entry

"Conservative root searching"

- Keep a stack that simulates the call stack
- Generate records for each parameter to procedure on entry
- "Parameter" records and "witness" records pushed to the stack

“Conservative root searching”

- Keep a stack that simulates the call stack
- Generate records for each parameter to procedure on entry
- “Parameter” records and “witness” records pushed to the stack
- When GC triggered, use everything on the stack as root

Extending ET2

- ET2 is **extensible** by design

Extending ET2

- ET2 is **extensible** by design
- One backend, multiple languages, multiple frontends

Extending ET2

- ET2 is **extensible** by design
- One backend, multiple languages, multiple frontends
- Same simple execution model

Why extend ET2?

- Support different languages

Why extend ET2?

- Support different languages
- Study memory use in functional programming languages
  - First-class functional closures
  - Lazy FPLs & thunks

Why extend ET2?

- Support different languages
- Study memory use in functional programming languages
  - First-class functional closures
  - Lazy FPLs & thunks
- Compare memory usage in different languages

Current state of ET2

- ET2/Java is mostly usable
- Some edge cases unimplemented (e.g. reflection)
- Around 10-100 times faster than Elephant Tracks
- ET2 backend is still work in progress (outside collaboration)

In the thesis...

- Details on the architecture & trace format
- Details on the algorithms described here
- More about implementation specifics

Thanks to my collaborators

Google: JC Beyler, Man Cao, Wessam Hassanein, Kathryn McKinley, Ryan Rose, Leandro Watanabe
ANU: Steve Blackburn
(all in alphabetical order)

and finally...

Thanks to my committee...

and finally...

Thanks to my committee...
and thanks to everyone who came today!

References I

Matthew Hertz, Stephen M Blackburn, J Eliot B Moss, Kathryn S. McKinley, and Darko Stefanović. 2002. Error-free Garbage Collection Traces: How to Cheat and Not Get Caught. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '02)*. ACM, New York, NY, USA, 140–151. https://doi.org/10.1145/511334.511352

Matthew Hertz, Stephen M. Blackburn, J. Eliot B. Moss, Kathryn S. McKinley, and Darko Stefanović. 2006. Generating Object Lifetime Traces with Merlin. *ACM Trans. Program. Lang. Syst.* 28, 3 (May 2006), 476–516. https://doi.org/10.1145/1133651.1133654

References II

Simon Holm Jensen, Manu Sridharan, Koushik Sen, and Satish Chandra. 2015. MemInsight: Platform-independent Memory Debugging for JavaScript. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 345–356. `https://doi.org/10.1145/2786805.2786860`

Luis Mastrangelo and Matthias Hauswirth. 2014. JNIF: Java Native Instrumentation Framework. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '14)*. ACM, New York, NY, USA, 194–199. `https://doi.org/10.1145/2647508.2647516`

References III

Nathan P. Ricci, Samuel Z. Guyer, and J. Eliot B. Moss. 2013. Elephant Tracks: Portable Production of Complete and Precise GC Traces. In *Proceedings of the 2013 International Symposium on Memory Management (ISMM '13)*. ACM, New York, NY, USA, 109–118. https://doi.org/10.1145/2464157.2466484

Raoul L. Veroy and Samuel Z. Guyer. 2017. Garbology: A Study of How Java Objects Die. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2017)*. ACM, New York, NY, USA, 168–179. https://doi.org/10.1145/3133850.3133854