Compiling the Web

Building a Just-in-Time Compiler for JavaScript

Andreas Gal
Mozilla Corporation

Computer Science Department
University of California, Irvine

Talk Outline

- Motivation
- Compiling Statically Typed vs Dynamically Typed Languages
- Trace Compilation and Trace Trees
- Implementation & Benchmarks
- Conclusions & Outlook

Motivation

- The Web is eating the Desktop's lunch.
  - The browser is becoming the next generation application platform.
  - GMail, Google Maps, Zimbra, Meebo
  - But also traditional "offline" applications: Google Docs, Google Spreadsheet, ...

Web Languages

- The Web uses a very different set of languages.
  - Desktop: Java, C, C++, C#, Delphi
  - Web Server: PHP, Python, Perl, Ruby, C/C++ (CGI), Java (JSP).
  - Browser: JavaScript, ActionScript (Flash), Java, C# (Silverlight), C/C++ (ActiveX).
- *Most dynamic web content is driven by dynamic languages.*

Trends

- Trends:
  - Client-side Java (slow, large footprint, fragmented) and ActiveX (security nightmare) are dying out.
  - Java seems to be holding up on the server side, but CGI is dying out (security!).
  - Microsoft is heavily investing in Silverlight, but adoption seems to be flat.

Observations

- Observations:
  - *Web programmers seem to like dynamic languages*.
  - Even evangelizing them (Silverlight!) seems to get you only so far. Despite superior tools and performance (Microsoft's C# Chess Demo) developers seem to prefer existing dynamic languages (JavaScript/PHP).
  - So what is the state of the art in JavaScript performance?

Lets travel back in time to June 17, 2008
(not even 6 month ago).

June 17, 2008

- Approximately 1.4bn (billion) Internet users [Miniwatts Marketing Group, January 2008].
- 95% of Internet users use a browser that supports JavaScript [W3Schools's web statistics, January 2008].
- *JavaScript is a purely interpreted language, no matter what browser you use.*
  - Only contenders: RHINO (JavaScript-to-JVM compiler), Tamarin (ActionScript).

August 23, 2008

- Mozilla releases the first Firefox build with a JavaScript Just-in-Time compiler (TraceMonkey).
  - Speedups in the range of 2x to 20x, depending on benchmark and application.
  - World wide press coverage, including a lot of non-technical venues (New York Times, Dallas Morning News, Der Standard).
  - *People seem to care.*

The Competition

- September 1, 2008
  - Google "slips out" Chrome.
  - Stellar on Google's own benchmarks, mixed bag on established benchmarks.
- September 18, 2008
  - Apple releases SquirrelFish Extreme (SFX).
  - Solid 1.5-2x speedup over already very fast interpreter plus RegExp JIT compiler.

October 19, 2008

- JavaScript performance has become the focal point of the new browser wars.
- Most non-Microsoft browsers will ship a JavaScript Just-in-Time compiler with the next release.
  - Even Microsoft is working on JavaScript performance. IE8 will feature a much faster JavaScript interpreter.

Key Questions

- Why did it take so long to get JIT compilation for JavaScript off the ground?
  - Java uses JIT compilation for over a decade.
  - HotSpot will turn 10 next year!
- State of the art very similar for Python, Ruby, PHP, Perl.
  - Some projects under way (PyPy, Parrot), but performance is still severely lacking.

Is compiling a dynamically typed language really that different from compiling a statically compiled language?

Compiling Dynamic Languages

- Lets take a look at machine code generation for Java:

```
int a, b, c;
...
c = a + b;
```

```
mov eax, [ebp+4] // a
add eax, [ebp+8] // b
mov [ebp+12], eax // c
```

Compiling Dynamic Languages

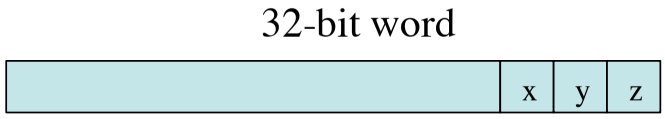
- What about JavaScript?

```
var a, b, c;
...
c = a + b;
```

```
if (isInt(a) && isInt(b) && !overflowsIfAdded(int(a), int(b)))
    c = box(int(a) + int(b));
else if (isDouble(a) && isDouble(b))
    c = box(double(a) + double(b))
else if (isString(a) && isString(b))
    c = box(concat(string(a), string(b)))
else
    c = handleReallyComplexCases(a, b); // hundreds of lines of code
```


Boxed Values

- Since all values are dynamically typed, the type has to be represented dynamically along with the value:



z = 1 31-bit integer
z = 0 object/string/boolean/double

Operators

- Almost all JavaScript operators have a different semantics based on the operand types.
  - 5-7 types, and for binary operators quite a few combinations!
- The implementation of the JSOP_ADD bytecode in SpiderMonkey consists of hundreds of bytes of machine code.
  - Compare that to 2-15 bytes for Java.

What about Objects?

- Object access in Java translates to straight forward machine code:

```
class X { public int a; };
X x = new X();
x.a = 1;
```

```
mov eax, 1
mov ebx, [ebp+4] // x
mov [ebx+offsetof(X,a)], eax
```

What about Objects?

- In JS, objects are collections of properties and use prototype-based inheritance:

```
x = { a: 1 }; // x=new Object(); x.a=1;
y = { a: 2 };
x.__proto__ = y;
print(x.a) => 1
delete(x.a);     // remove property
print(x.a) => 2 // prototype lookup
```


Prototype Chain

- For every "field" access, we have to potentially walk the entire prototype chain:

```
do {
    if (hasProperty(x, p))
        return getProperty(x, p);
    x = getPrototype(x);
} while (x != null);
```

Wait, it gets worse!

- Property names can be dynamic strings:

```
x = "y";
a[x + x] = 1;
print(a.yy) => 1
```

- In other words *offsetof* is a dynamic runtime function, not always a compile time constant.

What about Arrays?

- Arrays are almost as efficient as Objects in Java, except for an array bounds check:

```
int[] x = new int[3];
y = x[n];
mov ebx, [ebp+4] // x
mov eax, [ebp+8] // n
cmp eax, [ebx+0] // x.length
jnb @exception // if not below (unsigned)
mov eax, [ebx+eax*4+4]
```

Arrays in JavaScript

- Arrays in JavaScript look like Java, but they are really JavaScript objects (with all the quirks that come with that):

```
x = [1,2,3];
y = [4,5,6];
y.__proto__ = x;
print(y[1]) => 5; // zero-indexed
delete y[1];
print(y[1]) => 2; // not very intuitive
```


http://shaver.off.net/diary/

```
shaver@carnifex-4 5:src$ cat loop.js; Darwin_OPT.OBJ/js -j loop.js; time -p Darwin_OPT.OBJ/js -j loop.js
function main() { for (var i = 0; i < 300000000; i++) /* nothing */; }
main();
real 1.07
user 1.03
sys 0.03
shaver@carnifex-4 6:src$ cat loop.c; gcc -o loop loop.c; ./loop; time -p ./loop
int main() { int i; for (i = 0; i < 300000000; i++) /* nothing */; }
real 0.91
user 0.90
sys 0.00
```

http://shaver.off.net/diary/

```
shaver@carnifex-4 5:src$ cat loop.js; Darwin_OPT.OBJ/js -j loop.js; time -p Darwin_OPT.OBJ/js -j loop.js
function main() { for (var i = 0; i < 300000000; i++) /* nothing */; }
main();
real 1.07
user 1.03
sys 0.03
shaver@carnifex-4 6:src$ cat loop.c; gcc -o loop loop.c; ./loop; time -p ./loop
int main() { int i; for (i = 0; i < 300000000; i++) /* nothing */; }
real 0.91
user 0.90
sys 0.00
```

HOW ?

Translation Unit

- Compilers for statically typed languages traditionally use methods as translation unit.
  - Some compilers perform inlining, but the smallest translation unit is still an entire method.
- We can generate efficient machine code because at every point in a method the type of a value is fixed (no matter what the control flow does).
  - In case of Java for example, if *d* is declared as double, we can allocate it into a floating-point register and all arithmetic operations are emitted using floating-point math.

Static Typing

- Consider the following Java pseudo code:

```
double d = 0;
for (...) {
    if (cond) d = d + 0.5;
    else d = "";  // type error
}
return d + 1; // always a FP addition!
```

- *In Java, the control flow can affect values, but not types.*

Dynamic Typing

- In JavaScript, types are control-flow dependent:

```
var d = 0;
for (...) {
    if (cond) d = d + 0.5;
    else d = "";  // ok in JavaScript
}
return d + 1; // number or string "1"
```

Solutions

- Static Analysis
  - Some promising research work underway, but for non-trivial programs analysis time is seconds to minutes. User unlikely to be willing to wait that long.
- Generate generic code (similar to the corresponding interpreter implemenation).
  - Apple's Squirrelfish Extreme (SFX)
  - Google's V8 VM

Generating Generic Code

JavaScript: `for (var i = 0; i < 100; ++i) { /* do nothing */ }`

```
i = box(1)
label:
    if (isInt(i)) i = box(int(i) + 1);
    if (isDouble(i)) i = box(double(i) + 1.0);
    if (isString(i)) i = box(String2Num(string(i)) + 1.0);
    if (isBoolean(i)) i = box(Bool2Num(boolean(i)) + 1.0);
    if (isObject(i)) i = box(Object2Num(object(i)) + 1.0);
    if (isUndefined(i)) i = box(NaN);

    if (isInt(i) && (int(i) < 100)) goto label;
    if (isDouble(i) && (double(i) < 100.0)) goto label;
    if (isString(i) && (String2Num(i) < 100.0)) goto label;
    if (isBoolean(i) && (Bool2Num(i) < 100.0)) goto label;
    if (isObject(i) && (Object2Num(i) < 100.0)) goto label;
    if (isUndefined(i) && (NaN < 100.0)) goto label;
```

What we really want ...

JavaScript: `for (var i = 0; i < 100; ++i) { /* do nothing */ }`

```
i = box(1)
label:
    if (isInt(i)) i = box(int(i) + 1);
    if (isDouble(i)) i = box(double(i) + 1.0);
    if (isString(i)) i = box(String2Num(string(i)) + 1.0);
    if (isBoolean(i)) i = box(Bool2Num(boolean(i)) + 1.0);
    if (isObject(i)) i = box(Object2Num(object(i)) + 1.0);
    if (isUndefined(i)) i = box(NaN);

    if (isInt(i) && (int(i) < 100)) goto label;
    if (isDouble(i) && (double(i) < 100.0)) goto label;
    if (isString(i) && (String2Num(i) < 100.0)) goto label;
    if (isBoolean(i) && (Bool2Num(i) < 100.0)) goto label;
    if (isObject(i) && (Object2Num(i) < 100.0)) goto label;
    if (isUndefined(i) && (NaN < 100.0)) goto label;
```


Our Approach: Traces

- A trace represents the series of instructions executed by the CPU (or VM) while running a program.
  - While traces can be constructed statically [Trace Scheduling, Fisher, 1991], most often they are recorded dynamically. [Dynamo, Bala, 1999]
  - Conditional branches become side exit points.
  - Since loops dominate the execution time of most programs, we are are only interested in loop traces.

Example Trace

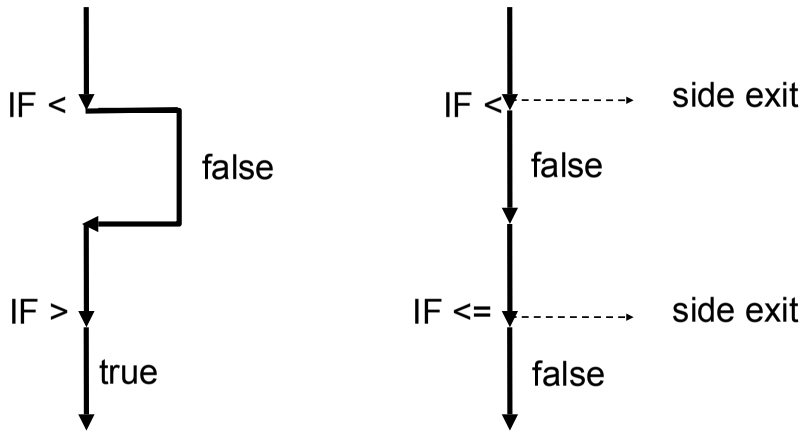
JavaScript: `for (var i = 0; i < 100; ++i) { /* do nothing */ }`

```
loop-header:
    if (int(i) >= 100)) break;
    i = box(int(i) + 1);
    goto loop-header;
```

- A trace starts at a loop header and loops back to it.
- A trace records the sequence of activities the VM executed while running the loop.
- If the VM operates on integers, we record a trace that operates on integers.

Control Flow

Conditional branches become Guard instructions.



Guard Instructions

- Traces are entered at the loop header, and execute until a guard instruction fails, in which case we return to the interpreter.
  - A trace is a speculative control-flow path.
- A trace is a linear sequence of instructions.
  - All instructions have **exactly one** predecessor instruction.
  - If the original trace conditions no longer hold, we must side-exit.

What about types?

JavaScript: `for (var i = 0; i < 100; ++i) { /* do nothing */ }`

```
loop-header:
    ensure(isInt(i));         // type guard
    ensure(int(i) < 100));    // loop condition guard

    ensure(isInt(i));         // type guard
    i = box(int(i) + 1);
    loop;                     // jump to loop header
```

- In addition to guarding the control-flow, we also have to guard the operand types if we only record a specialized instruction sequence.
- But wait ... we have a linear sequence of instructions ...

Type Stability on Trace

JavaScript: `for (var i = 0; i < 100; ++i) { /* do nothing */ }`

```
loop-header:
    ensure(isInt(i));        // type guard
    ensure(int(i) < 100));   // loop condition guard

    /* ensure(isInt(i)); */ // redundant
    i = box(int(i) + 1);
    loop;                    // jump to loop header
```

- Since traces specialize the control flow, the use of i always reads from the same definition of i.
  - The second guard is actually redundant.
- Next question: If types are stable, why box?

Unboxed Trace Code

JavaScript: `for (var i = 0; i < 100; ++i) { /* do nothing */ }`

```
loop-entry:
    ensure(isInt(i));  // check once when entering the trace
    ii = int(i);       // unbox when entering the trace
loop-header:
    ensure(ii < 100);  // loop condition guard
    ii = ii + 1;       // operate directly on unboxed value
    loop;              // jump to loop header
```

- Values are boxed only when we exit the trace (failed guard).
  - At compile-time, we store the types for each value at every side exit.

Summary: Traces

- A trace is a path through a loop in the original program.
- A trace is statically typed, but is only valid as long both control-flow and type-guards hold.
- Type-guards can often be hoisted out of the loop, producing a speculative, statically-typed equivalent representation of the original program.

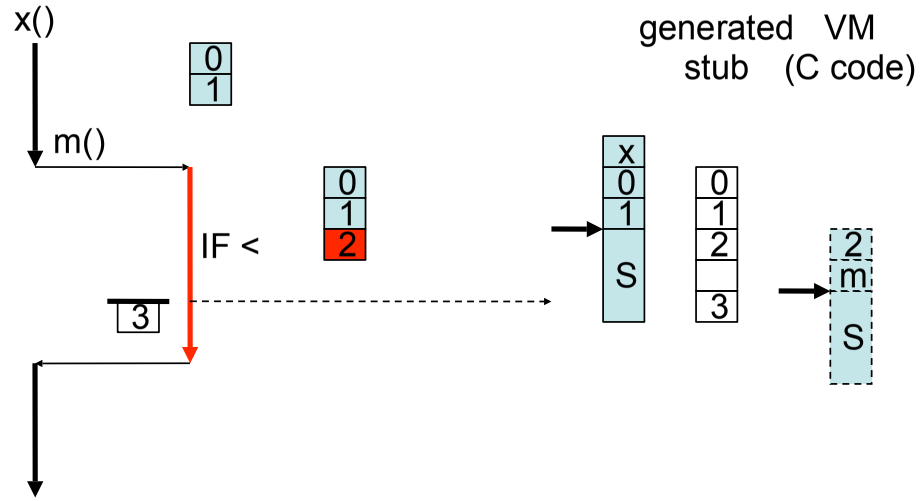
Inlining

- All instructions (potentially) affecting the control flow are replaced with guard instructions.
  - Conditional branches, switch tables, virtual method calls, etc.
- Code from invoked method is inlined into the trace. No method frames are created during trace execution, only when we side exit (if needed).

Recovery

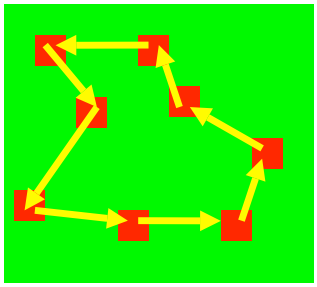
- Recovery stubs are generated for every side exit in the trace.
  - Compiler ensures that temporary values that have to be written back to the stack stay alive until the side exit.
  - Local variables modified in the trace are assumed to be live along all side exits (we can't and don't want to analyze code outside the trace).
  - Stack values and local variables are written back using the types stored by the compiler.
  - Execution returns to the interpreter, which will generate filler frames when returning from within a method call.

Deep Side Exits



Compiling Traces

SpiderMonkey

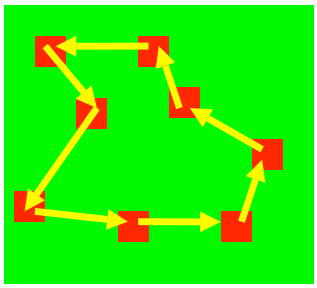


nanojit

Compiling Traces

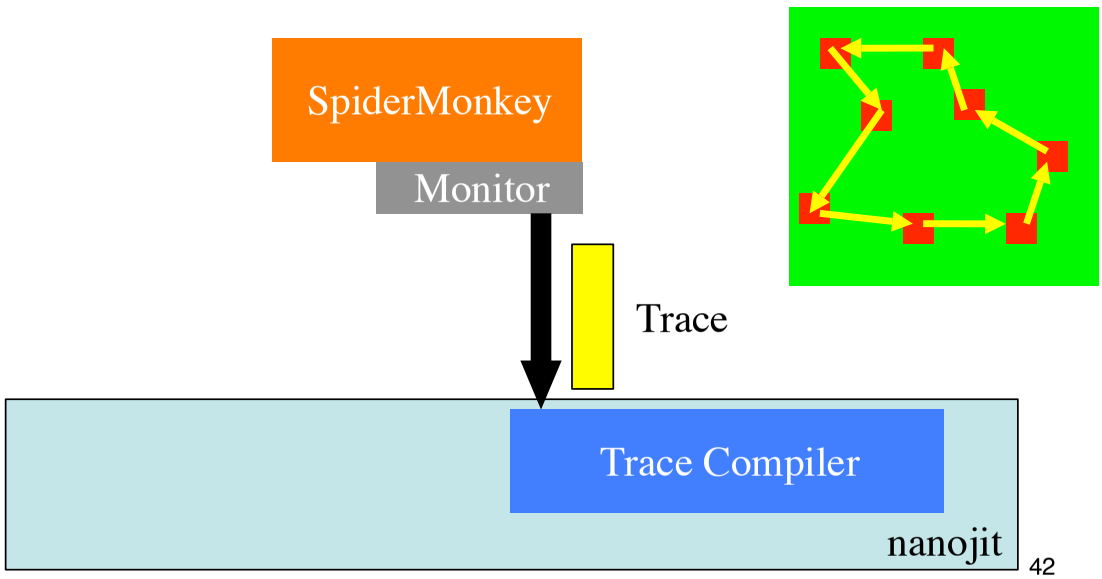
SpiderMonkey

Monitor



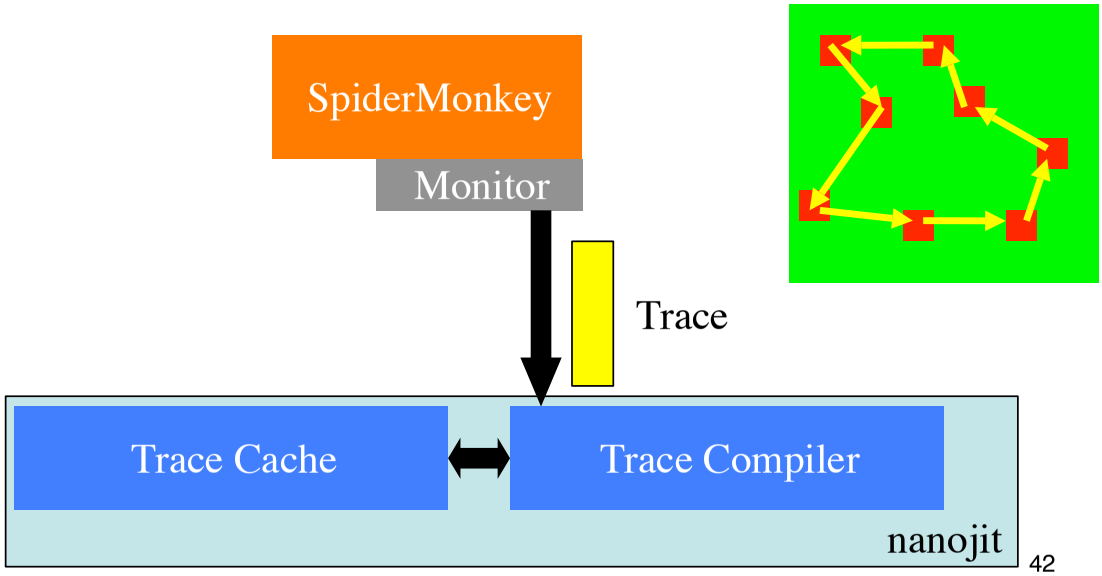
nanojit

Compiling Traces



42

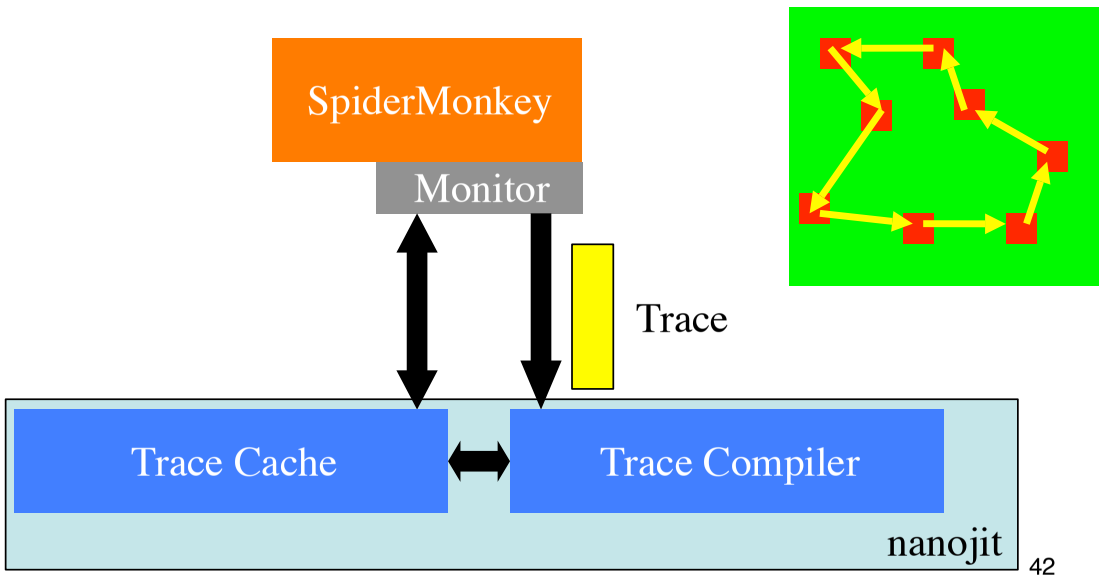
Compiling Traces



42

OOPSLA VMIL Workshop 2008
October 19, 2008, Nashville, Tennessee

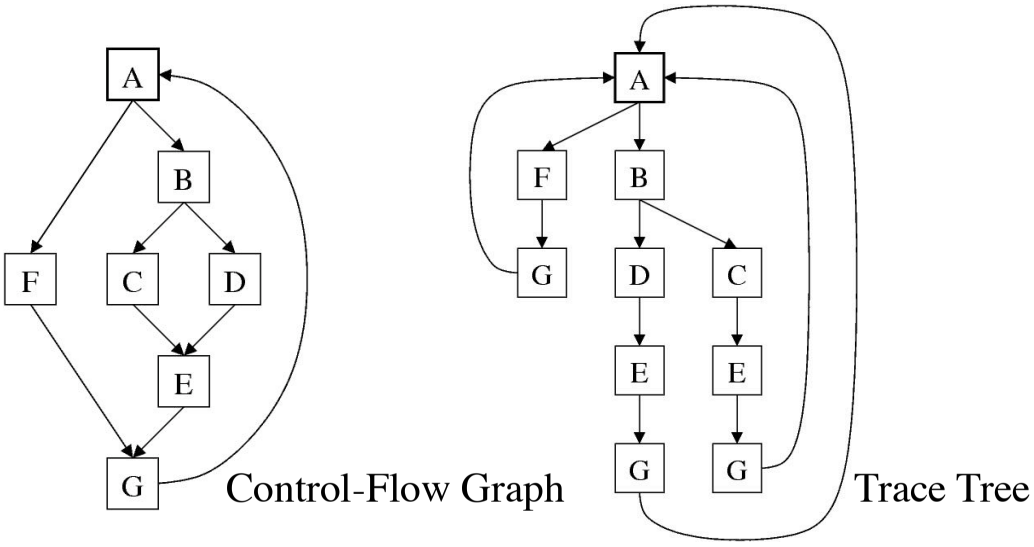
Compiling Traces



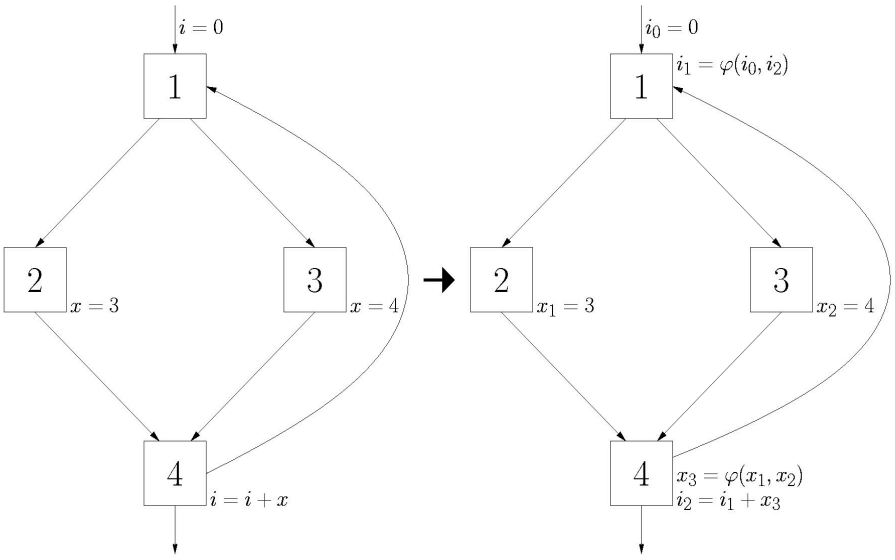
Trace Trees

- Record a new trace every time we bail out along a guard instruction.
- Stop recording when returning to the loop header or trace gets too long.
- The first trace is the *primary trace*.
- A *secondary trace* can start at any guard instruction and must also return to the loop header.

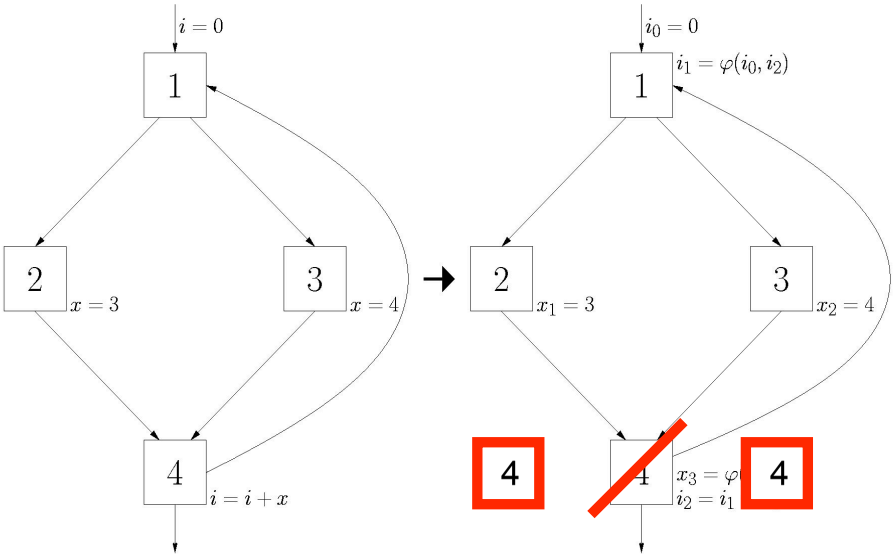
Trace Trees (2)



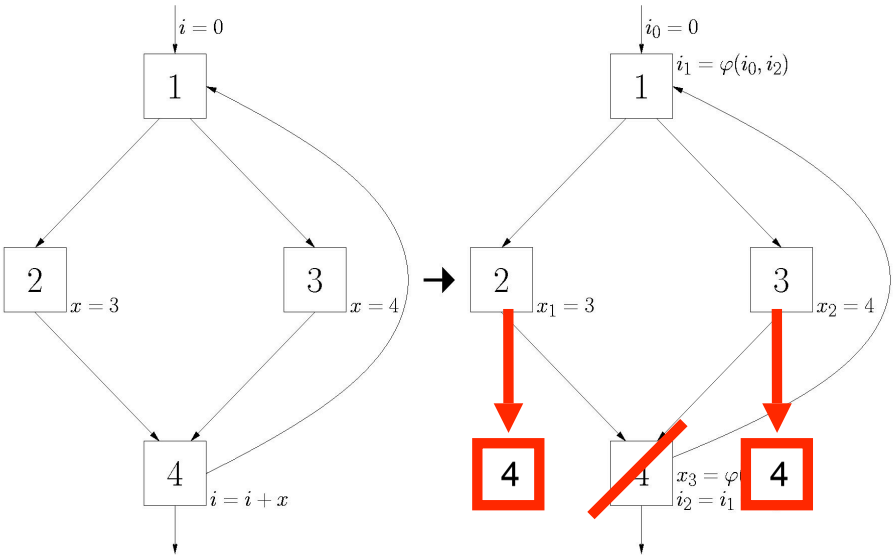
Tail Duplication



Tail Duplication



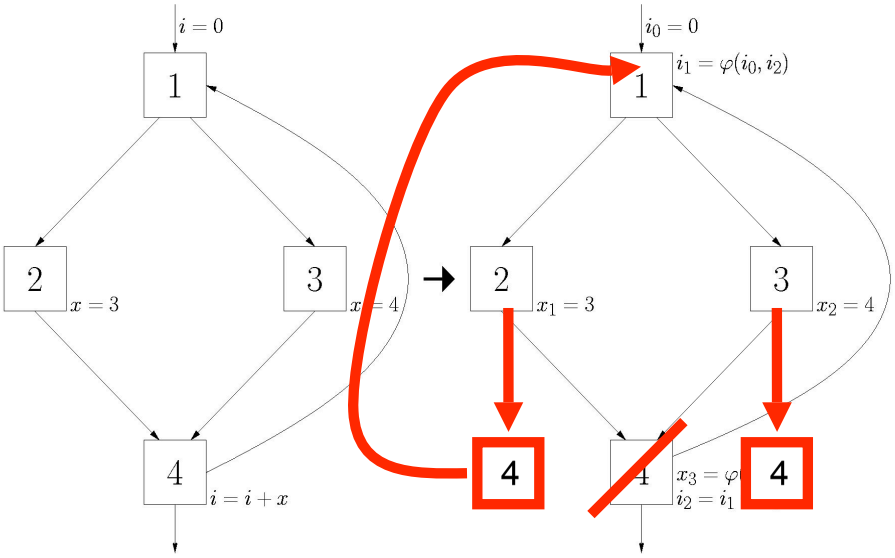
Tail Duplication



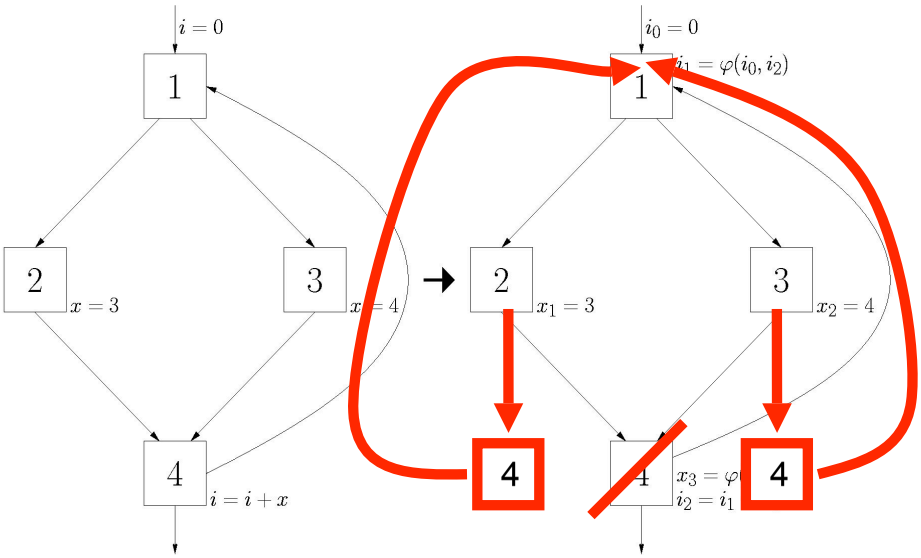
45

OOPSLA VMIL Workshop 2008
October 19, 2008, Nashville, Tennessee

Tail Duplication



Tail Duplication



Nested Trace Trees

- Each Trace Tree represents one loop. Trees can "call" inner loop trees.

```
for (var i = 0; i < 100; ++i)
    for (var j = 0; j < 100; ++j)
        ;
```

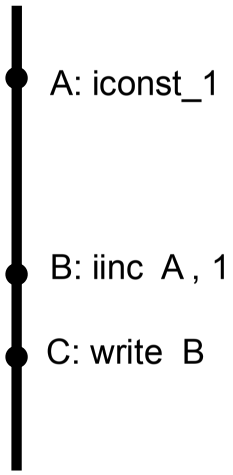
```
tree0-ii-is-int:
    guard(ii < 100)
    call tree1-jj-is-int
    ii++
    loop
```

```
tree1-jj-is-int:
    guard(jj < 100);
    jj++
    loop
```

Note that trees are specialized with respect to the types at loop entry.

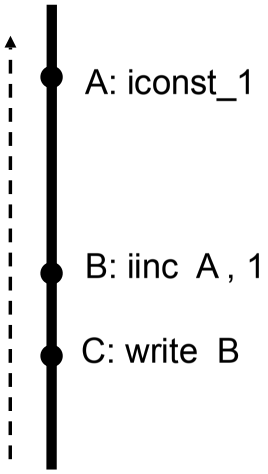
Compiling Traces

- Bottom-up code generation & register allocation.
  - No graph coloring!
- All uses are seen before the definition.
  - If no use assigned a register, definition is dead.
  - Deallocate register after emitting the defining instruction.



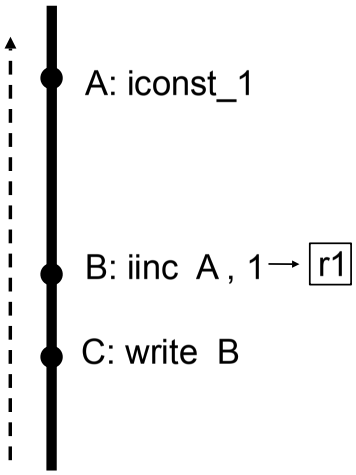
Compiling Traces

- Bottom-up code generation & register allocation.
  - No graph coloring!
- All uses are seen before the definition.
  - If no use assigned a register, definition is dead.
  - Deallocate register after emitting the defining instruction.



Compiling Traces

- Bottom-up code generation & register allocation.
  - No graph coloring!
- All uses are seen before the definition.
  - If no use assigned a register, definition is dead.
  - Deallocate register after emitting the defining instruction.

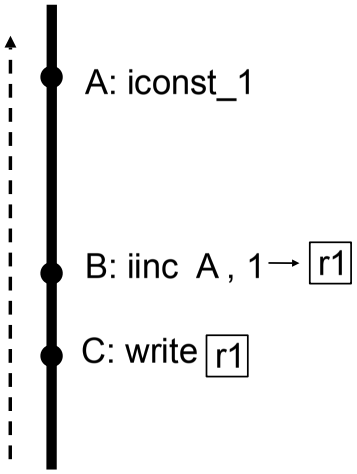


47

OOPSLA VMIL Workshop 2008
October 19, 2008, Nashville, Tennessee

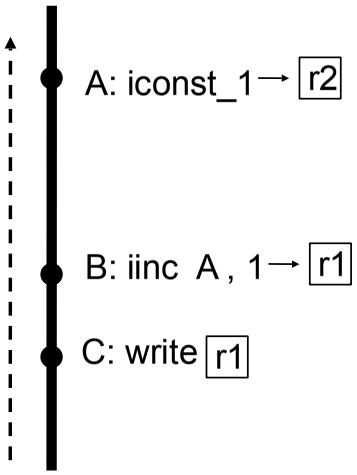
Compiling Traces

- Bottom-up code generation & register allocation.
  - No graph coloring!
- All uses are seen before the definition.
  - If no use assigned a register, definition is dead.
  - Deallocate register after emitting the defining instruction.



Compiling Traces

- Bottom-up code generation & register allocation.
  - No graph coloring!
- All uses are seen before the definition.
  - If no use assigned a register, definition is dead.
  - Deallocate register after emitting the defining instruction.

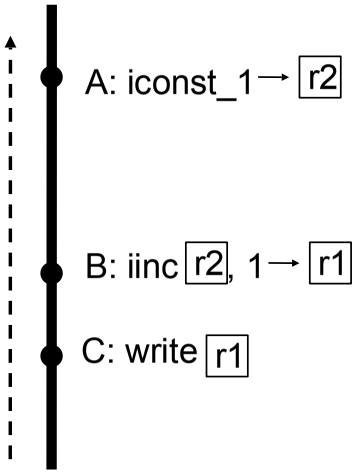


47

OOPSLA VMIL Workshop 2008
October 19, 2008, Nashville, Tennessee

Compiling Traces

- Bottom-up code generation & register allocation.
  - No graph coloring!
- All uses are seen before the definition.
  - If no use assigned a register, definition is dead.
  - Deallocate register after emitting the defining instruction.



Compiling Trace Trees

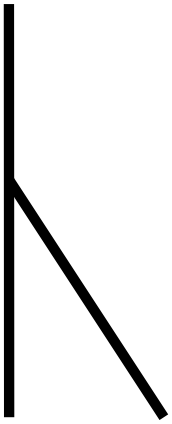
- The initial trace and its side-exit traces form a tree-shaped intermediate representation.
- Traces are compiled individually.
- How do we ensure correct register allocation?

Compiling Trace Trees

- The initial trace and its side-exit traces form a tree-shaped intermediate representation.
- Traces are compiled individually.
- How do we ensure correct register allocation?

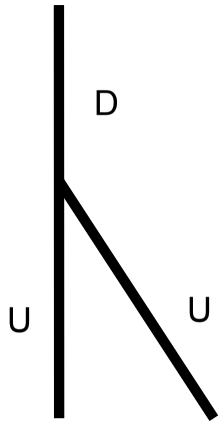
Compiling Trace Trees

- The initial trace and its side-exit traces form a tree-shaped intermediate representation.
- Traces are compiled individually.
- How do we ensure correct register allocation?



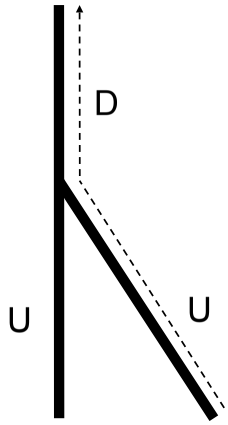
Compiling Trace Trees

- The initial trace and its side-exit traces form a tree-shaped intermediate representation.
- Traces are compiled individually.
- How do we ensure correct register allocation?



Compiling Trace Trees

- The initial trace and its side-exit traces form a tree-shaped intermediate representation.
- Traces are compiled individually.
- How do we ensure correct register allocation?

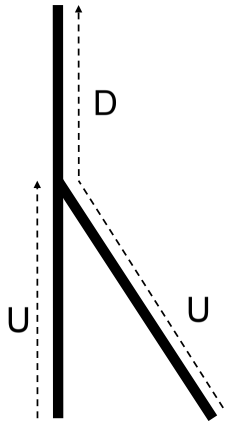


48

OOPSLA VMIL Workshop 2008
October 19, 2008, Nashville, Tennessee

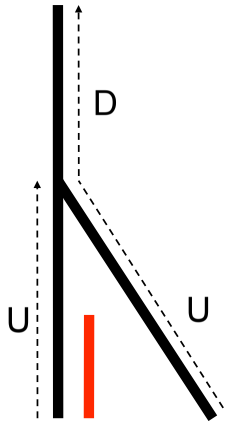
Compiling Trace Trees

- The initial trace and its side-exit traces form a tree-shaped intermediate representation.
- Traces are compiled individually.
- How do we ensure correct register allocation?



Compiling Trace Trees

- The initial trace and its side-exit traces form a tree-shaped intermediate representation.
- Traces are compiled individually.
- How do we ensure correct register allocation?



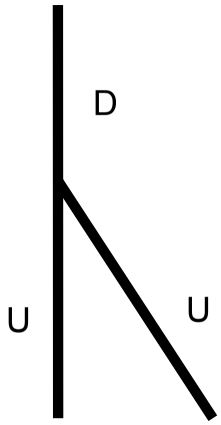
register no longer reserved

48

OOPSLA VMIL Workshop 2008
October 19, 2008, Nashville, Tennessee

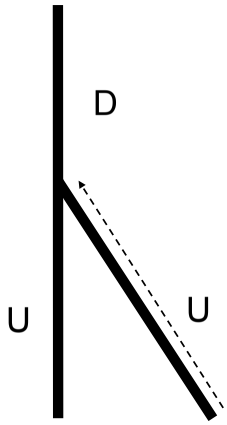
Trace Ordering

- *Always compile traces in reverse recording order.*
  - A trace can always only depend on traces that already exited before it.
  - By the time we compile a trace, all dependent traces (and thus all uses) were visited.



Trace Ordering

- *Always compile traces in reverse recording order.*
  - A trace can always only depend on traces that already exited before it.
  - By the time we compile a trace, all dependent traces (and thus all uses) were visited.

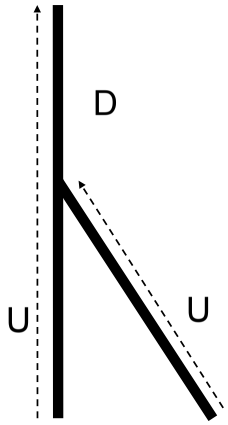


49

OOPSLA VMIL Workshop 2008
October 19, 2008, Nashville, Tennessee

Trace Ordering

- *Always compile traces in reverse recording order.*
  - A trace can always only depend on traces that already exited before it.
  - By the time we compile a trace, all dependent traces (and thus all uses) were visited.



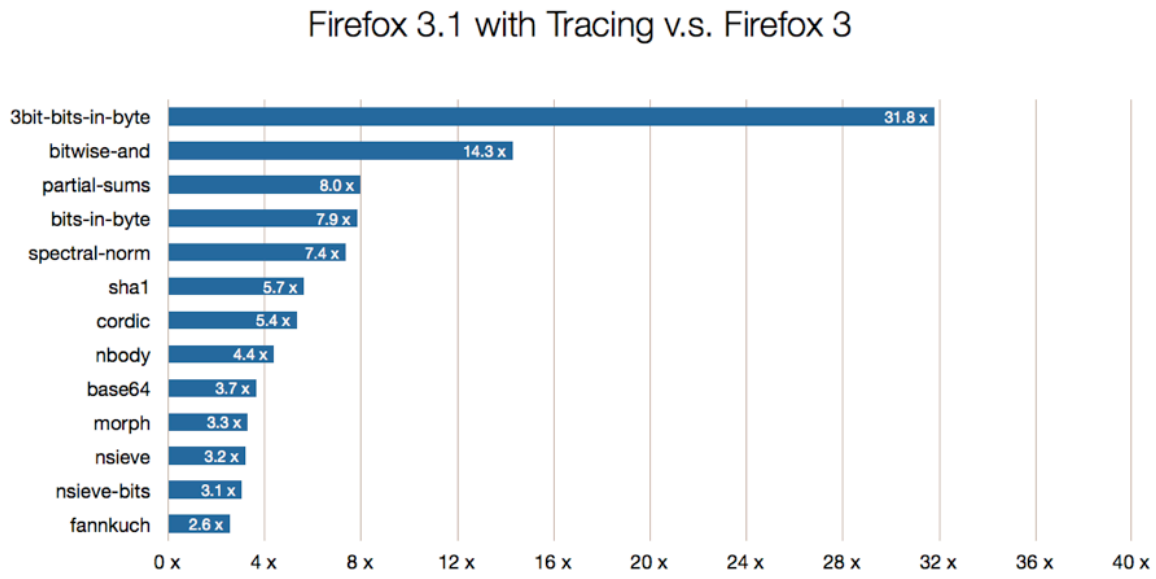
Benchmarketing

- JavaScript performance has only become the focal point of the "new browser war" fairly recently.
  - One somewhat established benchmark from Apple (SunSpider).
    - Some people think its not a very good abstraction of what matters on the web.
  - Some people prefer to bring their own benchmarks (V8 benchmark suite for example).

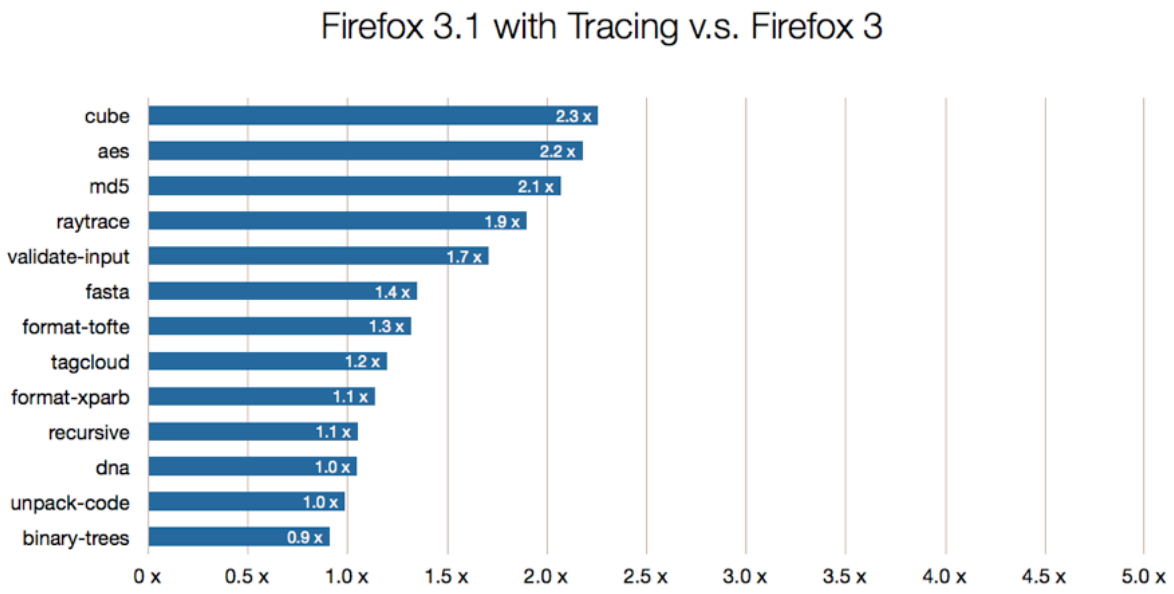
Benchmarks

- Highly volatile environment. Yesterday's numbers might be out of date today.
- The graphs I am about to show are meant to demonstrate the potential of trace-based compilation.
  - The numbers of SFX and V8 are for illustration purposes only ...

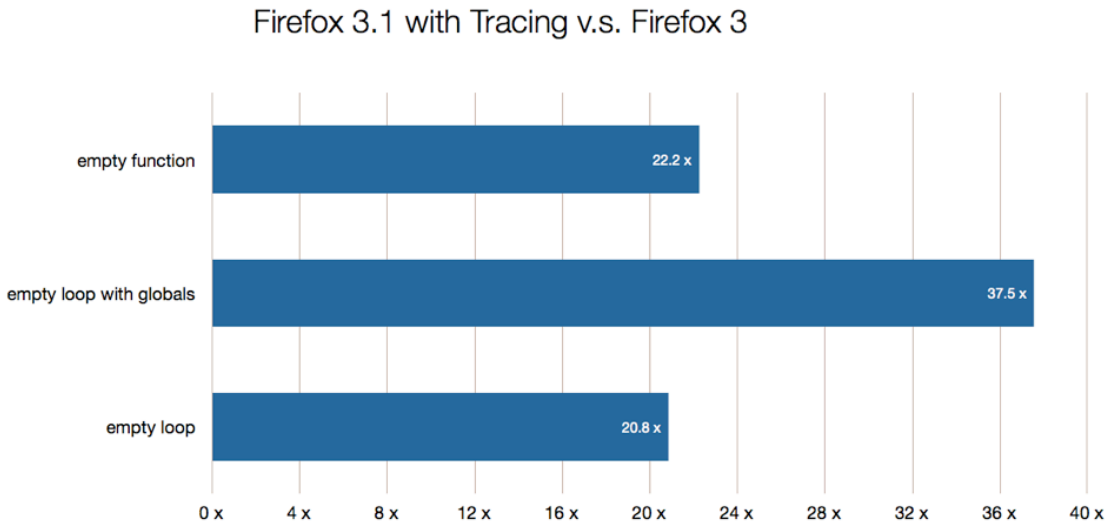
TraceMonkey vs Interpreter



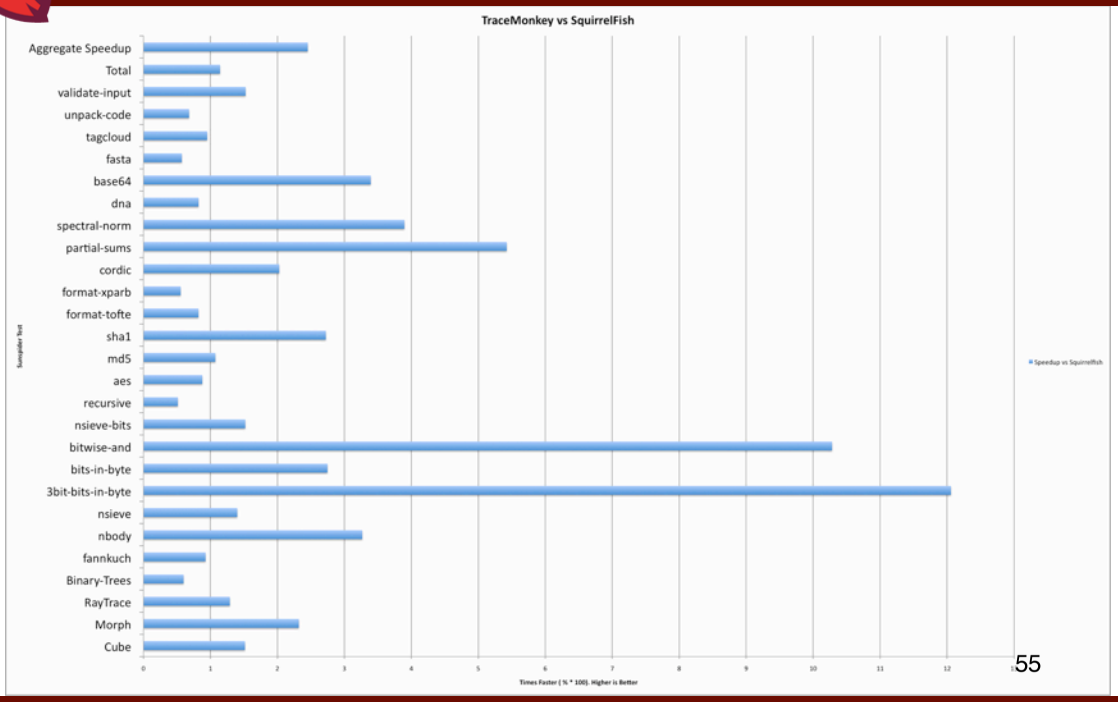
TraceMonkey vs Interpreter (2)



Microbenchmarks

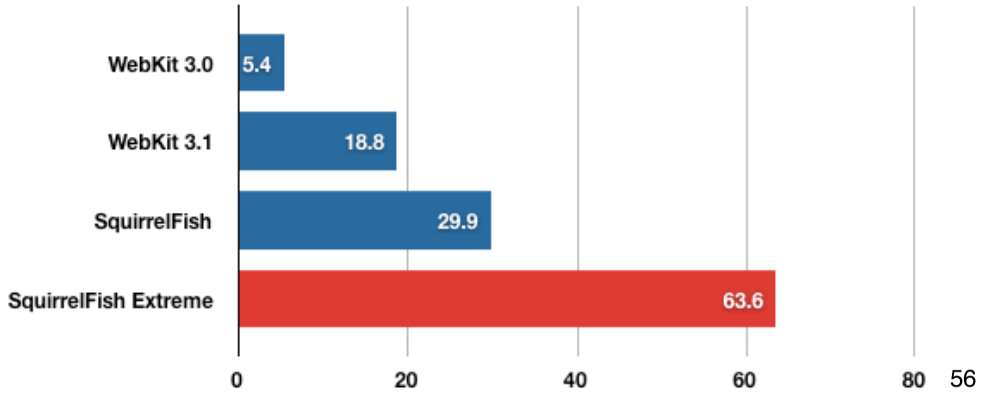


TraceMonkey vs SF



SF vs SFX

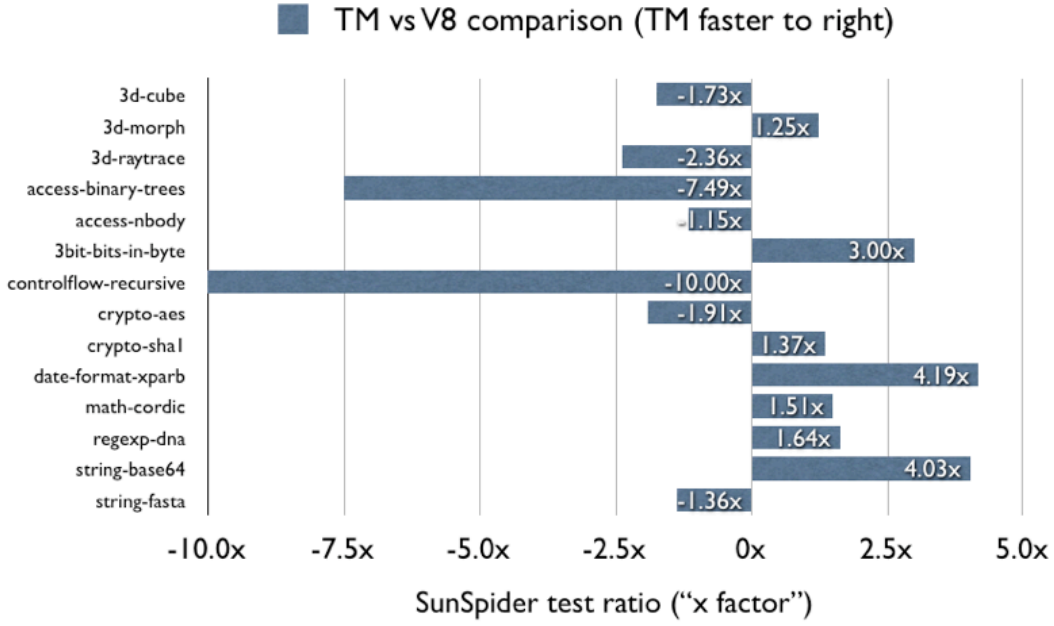
- SFX is about 2x faster than SF (including a 5x RegExp speedup which is not JS related.)



TM vs SF vs SFX

- SFX is up to 2x faster than SF's fast interpreter.
  - Highly efficient generic code generation (all types supported for every operation).
  - SFX's performance is likely the best you can do without specialization.
  - We achieve up to 12x speedup over SF using specialization. More potential for raw performance.

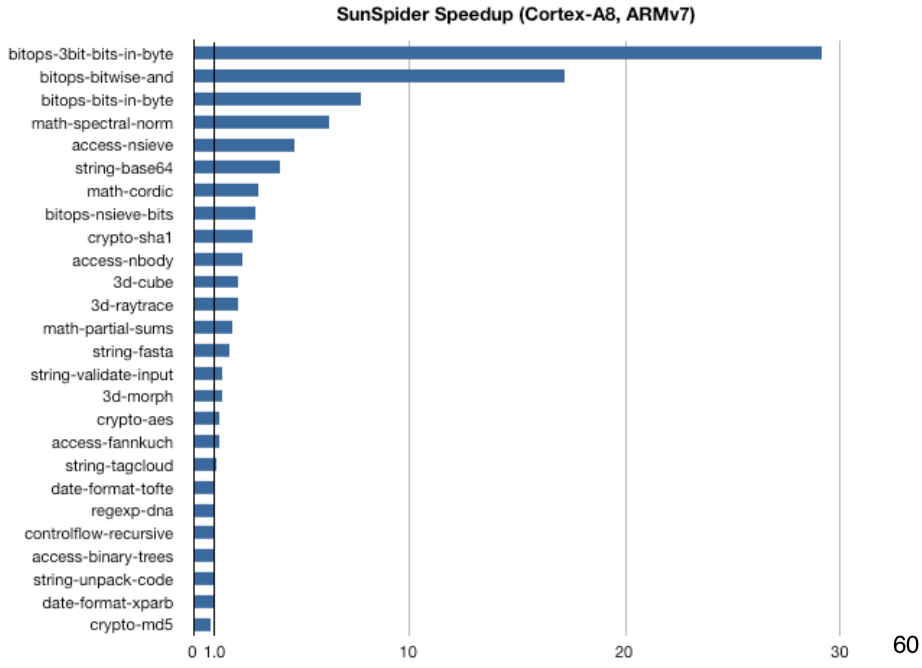
TraceMonkey vs V8



TM vs V8

- V8 is a brand-new, advanced JS engine.
  - Similar to SpiderMonkey's Property Cache, V8 uses inline caching to speed up property access ("Hidden Classes" in the V8 world).
  - However, all code is still generic. No type specialization. No inlining.
  - We are up to 4x faster on some benchmarks.
    - We don't trace recursion at all at the moment. Losing very badly there.

What about Mobile?



Mobile Devices

- Mobile devices are more constraint than Desktop system with respect to memory and CPU speed.
  - Specialized code means often much smaller machine code fragments.
  - Traces can be compiled very rapidly (compilation time often only a few micro seconds).

Conclusions

- A trace represents a path through a loop in the original program and is type-stable even for dynamically typed languages.
- Multiple paths through loops can be represented using trees of traces, and nested loops using nested trees of traces.
- Compiled trace code can in some cases compete with statically typed language code.

Conclusions (2)

- Trace Trees offer significantly higher peak performance than emitting method-based generic code.
  - The devil is in the details: tail duplication, code explosion.
- Very difficult to do comparative benchmarks at this point.
  - A lot more systematic research is needed. Remember, JS compilation is 6 month old!

Future Work

- Improve baseline performance of SpiderMonkey.
  - The performance of V8/SFX should be our baseline. Use traces to speed up suitable kernels.
- More aggressively avoid and compensate for tail duplication.
- Collaborate with other interested parties to produce a meaningful benchmark suite.

Questions?

Footnote: The speaker is looking for a tenure-track position in North America.