Kasper Lund, Software engineer at Google

Crankshaft

Turbocharging the next generation of web applications

Overview

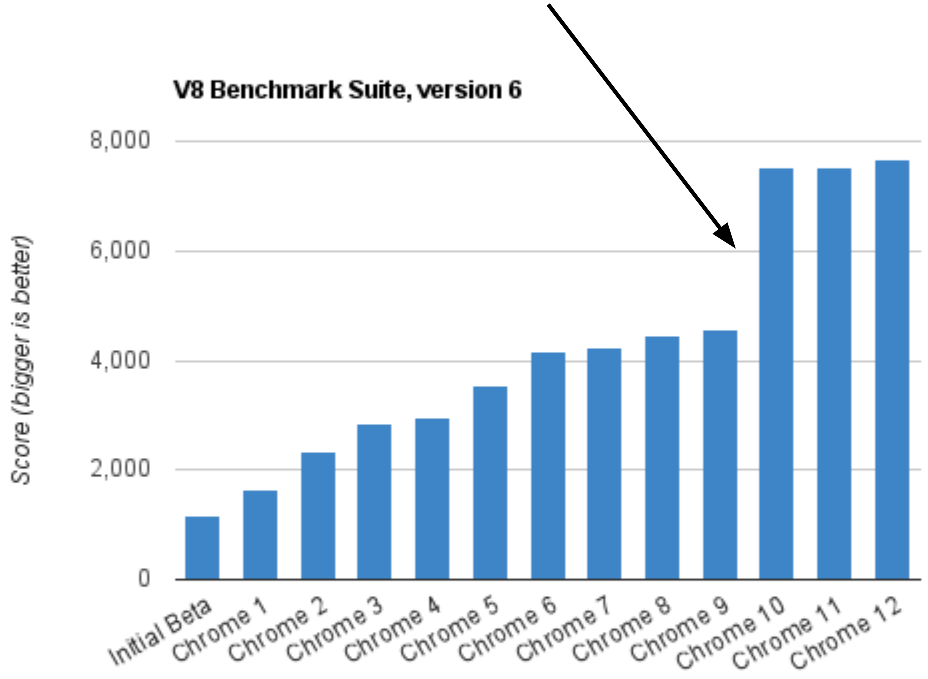
- Why did we introduce Crankshaft?
- Deciding when and what to optimize
- Type feedback and intermediate representation
- Deoptimization and on-stack replacement

Projects of interest

2010- *Google, Inc.*	**Dart**	Open-source programming language for the web
2006-2010 *Google, Inc.*	**V8**	Open-source, high-performance JavaScript
2002-2006 *Esmertec AG*	**OSVM**	Serviceable, embedded Smalltalk
2000-2002 *Sun Microsystems, Inc.*	**CLDC HI**	High-performance Java for limited devices

JavaScript performance is improving

Crankshaft introduced in Chrome 10: Adaptive optimizations driven by type-feedback



Motivation #1

Generated code kept increasing in size
and complexity

Code for optimized property access

Chrome 1 - code size is 14 bytes

```
function f(o) { return o.x; }
```

compiles to

```
push [ebp+0x8]         ;; push object
mov ecx,0xf712a885     ;; move key to ecx
call LoadIC            ;; call ic
```

Code for optimized property access

Chrome 6 - code size is 55 bytes

```
function f(o) { return o.x; }
```

compiles to

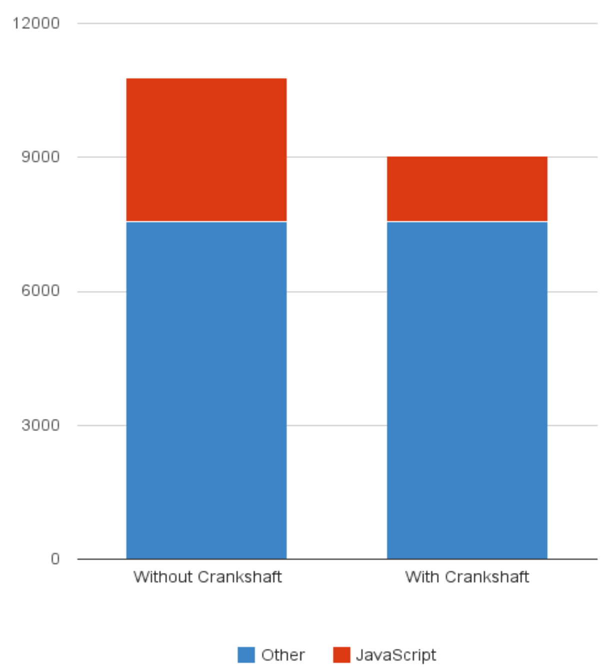
```
    mov eax,[ebp+0x8]            ;; load object
    test al,0x1                  ;; smi check object
    jz L1                        ;; go slow if not smi
    cmp [eax+0xff],0xf54d2021    ;; map check
    jnz L1                       ;; go slow if different map
L0: mov ebx,[eax+0xb]            ;; load property 'x'
    ...                          ;; return sequence
    ...
L1: mov ecx,0xf54db401           ;; move key to eax
    call LoadIC                  ;; call load ic
    test eax,0xffffffdb          ;; encoded offset of map check
    mov ebx,eax                  ;; shuffle around registers
    mov edi,[ebp+0xf8]           ;; reload function
    mov eax,[ebp+0x8]            ;; reload object
    jmp L0                       ;; jump to return
```

Motivation #2

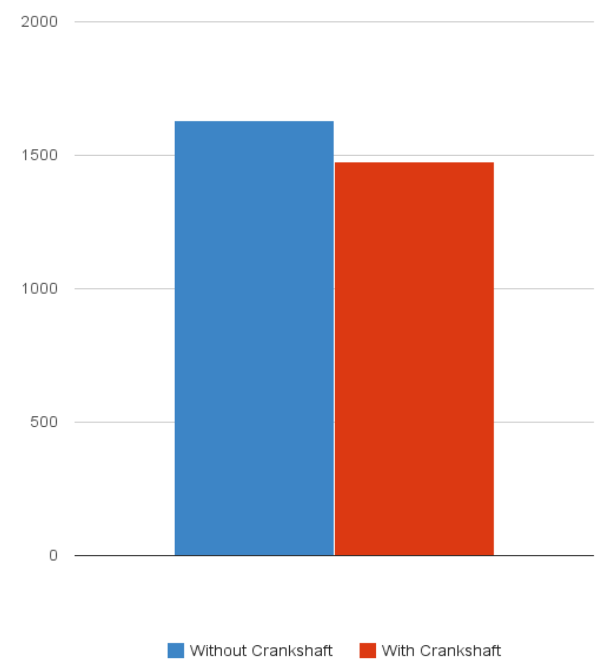
Spending time on optimizing everything led to slower web application startup

Adaptively optimizing helps startup time

Page cycler performance



Gmail startup performance



Motivation #3

Improving peak JavaScript performance required hoisting checks out of loops and doing aggressive method inlining

Example: Trivial loop with function call

```
function f() {
  for (var i = 0; i < 10000; i++) {
    for (var j = 0; j < 10000; j++) {
      g();
    }
  }
}

function g() {
  // Do nothing.
}
```

Generated code for inner loop of `f`

V8 version 2.5.9.22

```
L0: cmp esp,[0x8298a84]
    jc L3
    mov ecx,[esi+0x17]
    mov [ebp+0xf4],eax
    mov [ebp+0xf0],ebx
    push ecx
    mov ecx,0xf54047ed
    call 0xf53f5740
    mov esi,[ebp+0xfc]
    mov eax,[ebp+0xf0]
    add eax,0x2
    jo L2
    cmp eax, 0x4e20
    jnl L1
    mov ebx,eax
    mov eax,[ebp+0xf4]
    mov edi,[ebp+0xf8]
    jmp L0
L1: ...
L2: ...
L3: ...
```

V8 version 3.5.10.15 (optimized)

```
L0: cmp ebx,0x2710
    jnl L1
    cmp esp,[0x86595fc]
    jc L2
    add ebx,0x1
    jmp L0
L1: ...
L2: ...
```

Crankshaft

How does it actually work?

Crankshaft in one page

- Profiles and adaptively optimizes your applications
  - Dynamically recompiles and optimizes hot functions
  - Avoids spending time optimizing infrequently used parts

- Optimizes based on type feedback from previous runs of functions
  - No need to deal with all possible input value types
  - Generates specialized, compact code which runs fast

When and what should we optimize?

- Use statistical runtime profiling to gather information
  - Optimize when we are spending too much time in code we could speed up through aggressive optimizations

- Maintain sliding window of actively running JavaScript functions
  - Simulate a stack overflow every millisecond
  - Add samples for the top stack frames (with weights)

- Optimize functions that are *hot* in the sliding window on next invocation
  - Take size of the functions into account (only for large functions)
  - Start out optimizing less aggresively and then adjust thresholds

Trace from running the Richards benchmark

```
[marking Scheduler.schedule 0x3d1f643c for recompilation]
[optimizing: Scheduler.schedule / 3d1f643d - took 1.511 ms]
[marking runRichards 0x3d1f6130 for recompilation]
[optimizing: runRichards / 3d1f6131 - took 1.027 ms]
[marking DeviceTask.run 0x3d1f667c for recompilation]
[optimizing: DeviceTask.run / 3d1f667d - took 0.739 ms]
[marking Scheduler.suspendCurrent 0x3d1f64a8 for recompilation]
[marking HandlerTask.run 0x3d1f670c for recompilation]
[optimizing: HandlerTask.run / 3d1f670d - took 0.898 ms]
[marking Scheduler.queue 0x3d1f64cc for recompilation]
[optimizing: Scheduler.suspendCurrent / 3d1f64a9 - took 0.093 ms]
[optimizing: Scheduler.queue / 3d1f64cd - took 0.362 ms]
[marking WorkerTask.run 0x3d1f66c4 for recompilation]
[optimizing: WorkerTask.run / 3d1f66c5 - took 0.787 ms]
[marking TaskControlBlock.markAsNotHeld 0x3d1f6514 for recompilation]
[optimizing: TaskControlBlock.markAsNotHeld / 3d1f6515 - took 0.078 ms]
[marking Packet 0x3d1f622c for recompilation]
[optimizing: Packet / 3d1f622d - took 0.187 ms]
```

How does Crankshaft optimize?

- Classical optimizations
  - SSA-based high-level intermediate representation
  - Linear scan register allocation
  - Value range propagation
  - Global value numbering / loop-invariant code motion
  - Aggressive function inlining

- Novel approaches
  - Gathers type feedback from inline caches
  - Infers value representations (tagged, double, int32)

Optimizing based on type feedback

- Optimistically use the past to predict the future
  - Optimize based on assumptions about types
  - Guard optimized code patterns with assumption checks
  - Hoist expensive checks out of loops
- Aggressively inline field access, operations, and called methods
  - Avoid call overhead for "simple" operations
  - Preserve values in registers (less spills and restores)
  - Specialize target methods to the caller
- Improve arithmetic performance by avoiding to heap-allocate large integers and doubles (faster operations, less GC pressure)

Value representations

- Traditionally every value in V8 has been tagged
  - Tagged pointer to heap-allocated object
  - Tagged pointer to heap-allocated boxed double
  - Tagged small integer (31 bits)

- Crankshaft splits this into three separate representations
  - Tagged - generic tagged pointer (either of the above)
  - Double - IEEE 754 representation
  - Integer - 32 bit representation

- Increases the range of values we can represent as integers and avoids expensive boxing for doubles

Example (revisited)

```
function f() {
  for (var i = 0; i < 10000; i++) {
    for (var j = 0; j < 10000; j++) {
      g();
    }
  }
}

function g() {
  // Do nothing.
}
```

How do we optimize this?

Google™

Goal: No tagging, no overflow checks

```
L0: cmp ebx,0x2710
    jnl L1
    cmp esp,[0x86595fc]
    jc L2
    add ebx,0x1
    jmp L0
L1: ...
L2: ...
```

Generated code for inner loop of `f`

V8 version 2.5.9.22

```
L0: cmp esp,[0x8298a84]
    jc L3
    mov ecx,[esi+0x17]
    mov [ebp+0xf4],eax
    mov [ebp+0xf0],ebx
    push ecx
    mov ecx,0xf54047ed
    call 0xf53f5740 ;; code: CALL_IC
    mov esi,[ebp+0xfc]
    mov eax,[ebp+0xf0]
    add eax,0x2
    jo L2
    cmp eax, 0x4e20
    jnl L1
    mov ebx,eax
    mov eax,[ebp+0xf4]
    mov edi,[ebp+0xf8]
    jmp L0
L1: ...
L2: ...
L3: ...
```

V8 version 3.5.10.15 (unoptimized)

```
L0: push [esi+0x13]
    mov ecx,0x5b117639
    call 0x2f6eb2c0 ;; code: CALL_IC
    mov esi,[ebp+0xfc]
    mov eax,[ebp+0xf0]
    test al,0x1
    jz L1
    ...
L1: add eax,0x2
    jo L2
    test al,0x1
    jc L3
L2: ...
L3: mov [ebp+0xf0],eax
    cmp esp,[0x85eb5fc]
    jnc L4
    ...
L4: push [ebp+0xf0]
    mov eax,0x4e20
    pop edx
    mov ecx,edx
    or ecx,eax
    test cl,0x1
    jnc L5
    cmp edx,eax
    jl L0
L5: ...
```

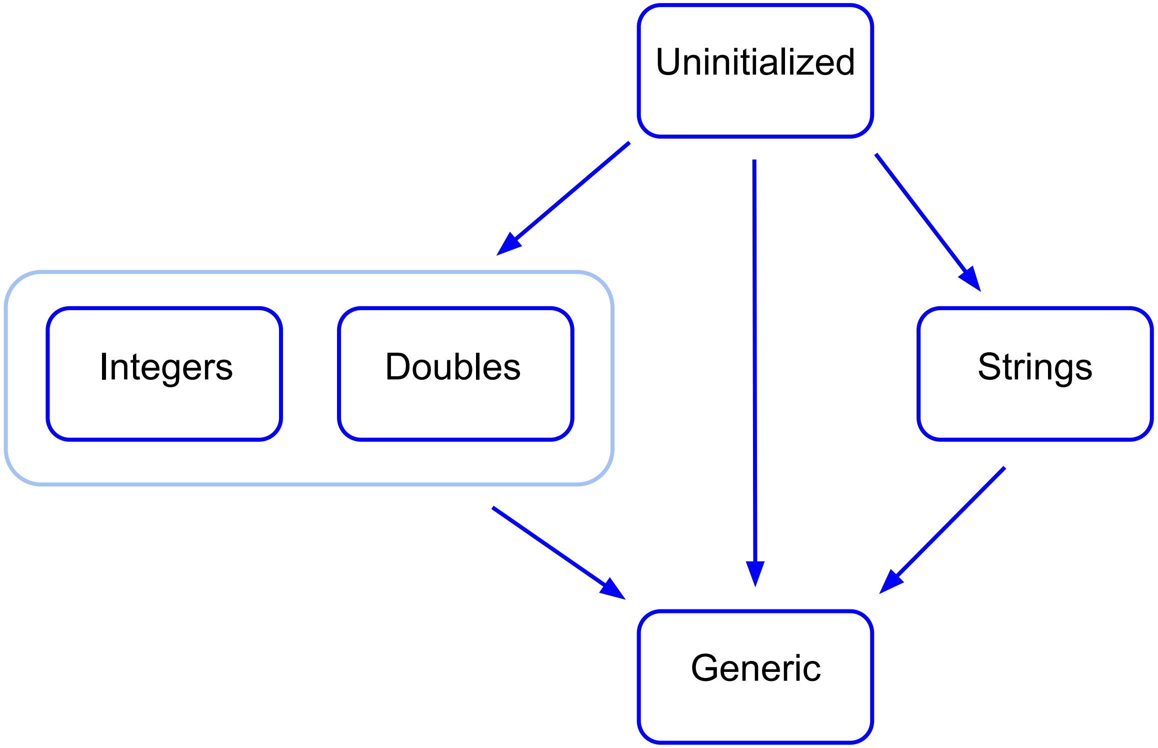
Instructions for computing `j + 1`

Capturing type feedback

```
     ...
     add eax,0x2
     jo L2
     test al,0x1
     jc L3
L2:  sub eax,0x2
     mov edx,eax
     mov eax,0x2
     call 0x2f6da520
     test al,0x11
L3:  ...
```

Call to binary operation stub
(rewritten on demand)

Binary operation states



High-level intermediate representation

```
function f(x, y) { return x + y; }
```

```
B0:
0 v0  block entry
1 t2  parameter 0 ; this
2 t3  parameter 1 ; x
2 t4  parameter 2 ; y
0 v8  simulate id=6  var[0] = t2 var[1] = t3 var[2] = t4
0 v9  goto B1

B1:
0 v5  block entry
1 i6  add t3 t4 !
0 v7  return i6
```

Introduce explicit `change` instructions

```
function f(x, y) { return x + y; }
```

B0:

```
0 v0  block entry
1 t2  parameter 0 ; this
2 t3  parameter 1 ; x
2 t4  parameter 2 ; y
0 v8  simulate id=6  var[0] = t2 var[1] = t3 var[2] = t4
0 v9  goto B1
```

B1:

```
0 v5  block entry
1 i10 change t3 t to i
1 i11 change t4 t to i
1 i6  add i10 i11
1 t12 change i6 i to t
0 v7  return t12
```

Adding strings instead of integers

```
function f(x, y) { return x + y; }
```

B0:

```
0 v0  block entry
1 t2  parameter 0 ; this
2 t3  parameter 1 ; x
2 t4  parameter 2 ; y
0 v9  simulate id=6  var[0] = t2 var[1] = t3 var[2] = t4
0 v10 goto B1
```

B1:

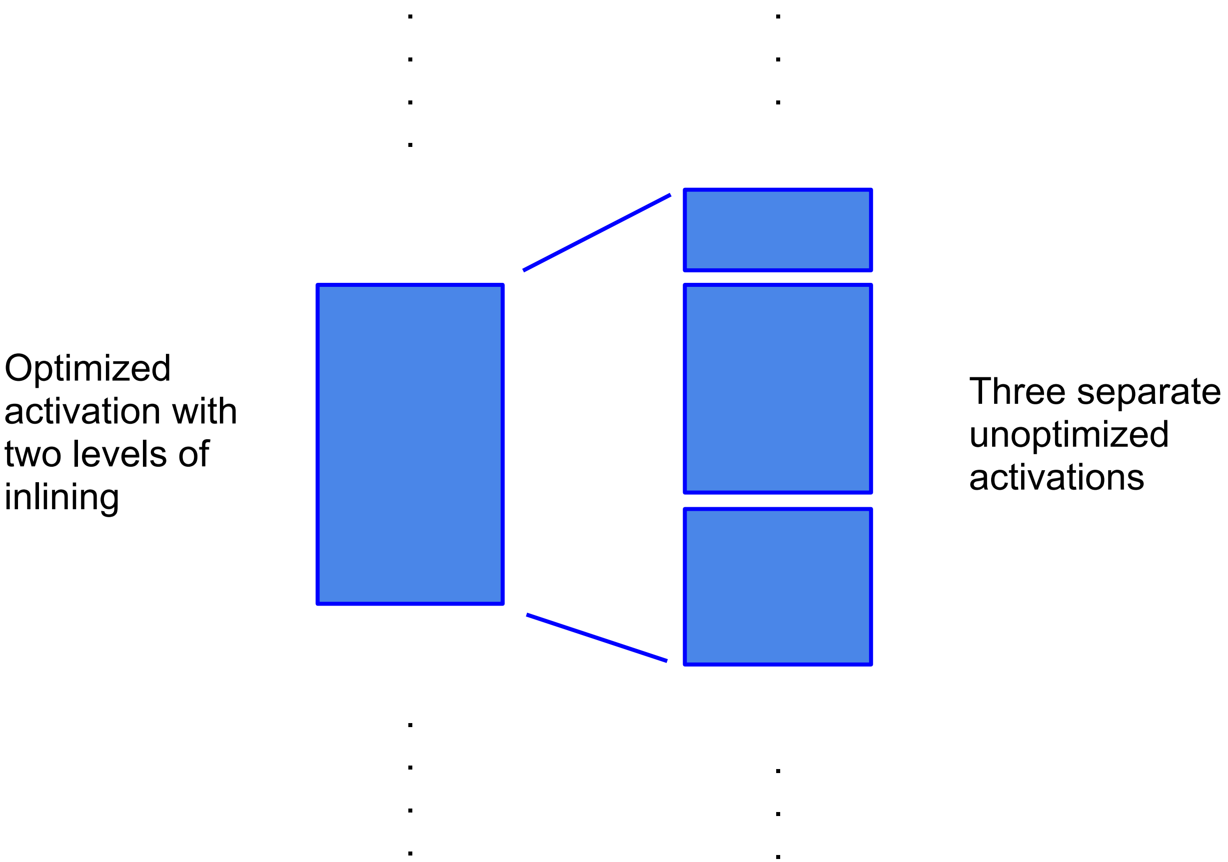
```
0 v5  block entry
0 t6  add* t3 t4 !
0 v7  simulate id=4  push t6
0 v8  return t6
```

The real key: Deoptimization

- Deoptimization lets us bail out of optimized code
  - Handle uncommon cases in unoptimized code
  - Support debugging without slow downs

- Must convert optimized activations to unoptimized ones
  - Map stack slots and registers to other stack slots
  - Update return address, frame pointer, etc
  - Box int32 and double values that are not valid smis
  - Allocate the "arguments object" if necessary

Deoptimization (continued)



Google™

On-stack replacement

- The runtime profiler marks functions for recompilation but do not recompile them before they are re-entered
  - If your application or benchmark consists only of a single function invocation we never get to optimize

- On-stack replacement is the opposite of deoptimization
  - Replaces unoptimized activations with the equivalent optimized versions and sets up register state
  - Allows optimizing functions while they are running in tight loops which mostly makes sense for benchmarks

- On-stack replacement happens at backward branches
  - Piggy backs on the stack overflow check
  - We prefer to do on-stack replacement in outer loops

Final remarks

- JavaScript performance has improved a lot over the last years
  - Lots of competitive pressure (great for the users)
  - Other vendors are experimenting with SSA-based compilation

- If you write your program in the right subset of JavaScript, there is a very good chance it will perform really, really well

- ... but hitting the JavaScript performance sweet spot is not trivial
  - Make use of profiling to figure out where your app spends its time
  - File performance bugs (we love new benchmarks)

Thank you for listening

Any questions?