Optimizing ActionScript Bytecode using LLVM

10/2/2009

Scott Petersen

Adobe Systems, Inc.

ActionScript 3

- Adobe Flash/AIR app development language

- EcmaScript based – "JavaScript with classes and types"
  - var x; // implicitly a variant – JS style
  - var x:int; // x is an int!
  - var x:*; // explicitly a variant

- ActionScript Bytecode (ABC) reminiscent of JVM bytecode
  - Verified
  - Stack oriented
  - Object oriented

ActionScript 3

- JIT compiler + interpreter + garbage collector + basic support library in around 1.5M uncompressed, 600k compressed for x86
  - Open Source Tamarin Project http://www.mozilla.org/projects/tamarin
- Straightforward AS3 compiler
  - Effectively non-optimizing

ActionScript 3 Performance

- Performance for AS => AS3 compiler + Tamarin JIT
  - Roughly 1.2% of native optimized C (Scimark2 numeric benchmark)
- Performance for C => C/LLVM based frontend (Alchemy) + Tamarin JIT
  - Roughly 30% of native optimized C (Scimark2 numeric benchmark)
- Performance for Java => javac + JRE 6
  - Roughly 60% of native optimized C (Scimark2 numeric benchmark)

ActionScript 3 Performance

- C code running on the Tamarin JIT is >20x faster than AS3!
  - Why is C code so fast on Tamarin?
  - Why is AS3 code so slow on Tamarin?

- Alchemy generated code
  - Avoids some known performance pitfalls in Tamarin
    - AS3 has a variant type – boxing and unboxing is expensive – Alchemy avoids this
    - AS3 often does many object allocations, taxing the GC – Alchemy uses a single "ram" object with fast access opcodes
    - Tamarin's parameter passing can be inefficient – Alchemy uses a virtual stack
    - Alchemy uses almost no dynamic property access, calling, etc.
  - Takes advantage of LLVM's aggressive optimization capabilities

ActionScript3 + LLVM?

- Could AS3 take advantage of LLVM's optimizations?
  - Some optimizations are not applicable
    - Memory/pointer specific
  - Some are
    - Loop transforms
    - Data flow
    - Arithmetic
    - DCE
    - Inlining! – but not for a large class of call types in AS3…
- LLVM doesn't understand AS3 or ABC!

ActionScript3 + LLVM?

- Alchemy in reverse
  - Instead of C => LLVM BC => (AS3 =>) ABC…
  - (AS3 =>) ABC => LLVM BC => ABC
- Generate an SSA representation of ABC
  - Open source tool "GlobalOptimizer" written by Adobe/Tamarin developer Edwin Smith already does this!
  - And does ABC specific type analysis, SCCP, DCE, etc.
- Convert SSA rep to / from LLVM
  - Generated bitcode does NOT have to be "real": we never generate platform assembly
  - opt!
- Reconstruct ABC from SSA rep
  - GlobalOptimizer

ActionScript3 + LLVM?

- Invent types for non-simple AS3 values
  - Strings, objects, variants become LLVM opaque type
- Generate an LLVM function for each AS3 function in a given ABC
- Convert most ABC opcodes to CallInst calls to placeholder functions
  - i.e., ABC opcode newobject =>
    - %1 = call avm2val avm2_newobject(…)
  - On placeholder functions, set memory side effect characteristics to allow LLVM some freedom
- Convert ABC flow control to appropriate LLVM instructions
  - jump L1 =>
    - br label %L1

ActionScript3 + LLVM?

- Convert arithmetic to appropriate LLVM instructions
  - i.e., ABC opcode add_i =>
    - %3 = call i32 @avm2unbox_i32(avm2val %1)
    - %4 = call i32 @avm2unbox_i32(avm2val %2)
    - %5 = add i32 %3, %4
    - %6 = call avm2val @avm2box_i32 (i32 %5)
  - Can use type info gleaned by GlobalOptimizer
- Convert statically known calls (i.e., callstatic) to CallInsts
  - callstatic CopyMatrix =>
    - call avm2val @CopyMatrix(avm2val %1, avm2val %2, avm2val %3)
- Eliminate redundant boxing/unboxing
  - box(unbox(x)) => x
  - unbox(box(x)) => x

ActionScript3 + LLVM?

- Simple AS3 function

```
function CopyMatrix(B:Array, A:Array):void
{
        var M:uint = A.length;
        var N:uint = A[0].length;

        var remainder:uint = N & 3;                         // N mod 4;

        for (var i:uint=0; i<M; i++)
        {
            var Bi:Array = B[i];
            var Ai:Array = A[i];
            for (var j:uint=0; j<remainder; j++)
                Bi[j] = Ai[j];
            for (j=remainder; j<N; j+=4)
            {
                Bi[j] = Ai[j];
                Bi[j+1] = Ai[j+1];
                Bi[j+2] = Ai[j+2];
                Bi[j+3] = Ai[j+3];
            }
    }
}
```

ActionScript3 + LLVM?

- As ABC

```
function CopyMatrix(Array,Array):void                    /* disp_id=45 method_id=0 */
{
  // local_count=10 max_scope=1 max_stack=5 code_len=210
  0       getlocal0
  1       pushscope
  2       pushnull
  3       coerce                  Array
  5       setlocal                7
  7       pushnull
  8       coerce                  Array
  10      setlocal                8
  12      pushbyte                0
  14      convert_u
  15      setlocal                9
  17      getlocal2
  18      getproperty             length
  20      convert_u
  21      setlocal3
  22      getlocal2
  23      pushbyte                0
  25      getproperty             null
  27      getproperty             length
  29      convert_u
  30      setlocal                4
  32      getlocal                4
  34      pushbyte                3
  36      bitand
  37      convert_u
  38      setlocal                5
  40      pushbyte                0
  42      convert_u
  43      setlocal                6
  45      jump                    L1
```

ActionScript3 + LLVM?

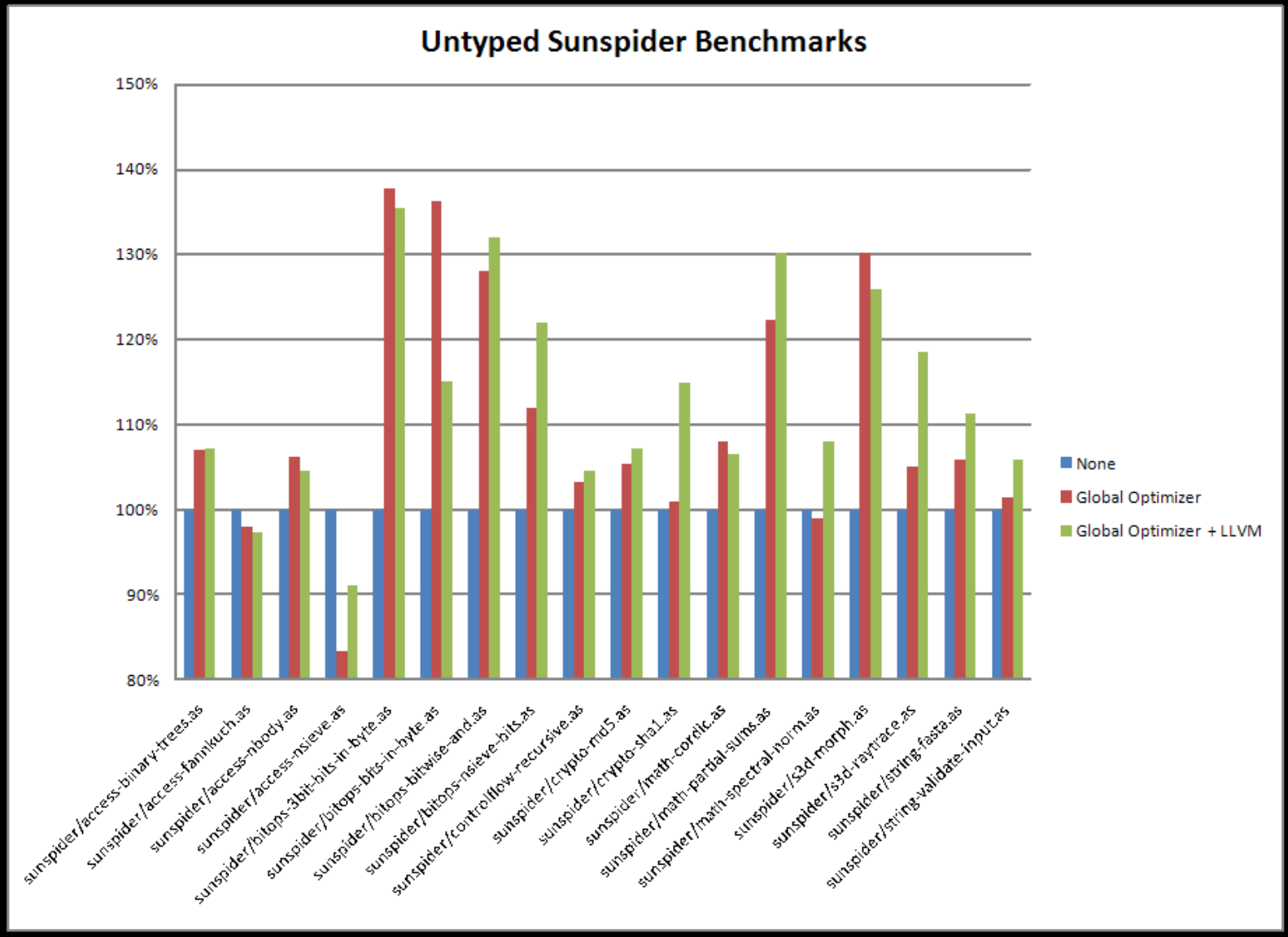
- As BC

```
; ModuleID = 'SparseCompRow'

declare avm2val @avm2_getproperty(...) readonly
declare void @avm2_setproperty(...)
declare avm2val @avm2_coerce(...) readnone

define avm2val @GO_m6_CopyMatrix(avm2val, avm2val, avm2val) {
bb_m6_b0_0_:
        %i = call avm2val (...)* @avm2_getproperty( avm2val %2, avm2ref bitcast (i32 24 to avm2ref) )           ; <avm2val> [#uses=1]
        %i41 = add i32 0, 0             ; <i32> [#uses=1]
        %i42 = call avm2val @avm2box_i32( i32 %i41 )            ; <avm2val> [#uses=1]
        %i1 = call avm2val (...)* @avm2_pushbyte( i32 0 )               ; <avm2val> [#uses=0]
        %i2 = call avm2val (...)* @avm2_getproperty( avm2val %2, avm2val %i42, avm2ref bitcast (i32 5 to avm2ref) )            ; <avm2val> [#uses=1]
        %i3 = call avm2val (...)* @avm2_getproperty( avm2val %i2, avm2ref bitcast (i32 58 to avm2ref) )         ; <avm2val> [#uses=1]
        %i4 = call avm2val (...)* @avm2_convert_u( avm2val %i3 )                ; <avm2val> [#uses=3]
        %i43 = add i32 3, 0             ; <i32> [#uses=2]
        %i44 = call avm2val @avm2box_i32( i32 %i43 )            ; <avm2val> [#uses=6]
        %i88 = call double @avm2unbox_double( avm2val %i44 )            ; <double> [#uses=1]
        %i84 = call double @avm2unbox_double( avm2val %i44 )            ; <double> [#uses=1]
        %i5 = call avm2val (...)* @avm2_pushbyte( i32 3 )               ; <avm2val> [#uses=0]
        %i45 = call i32 @avm2unbox_i32( avm2val %i4 )           ; <i32> [#uses=1]
        %i46 = call i32 @avm2unbox_i32( avm2val %i44 )          ; <i32> [#uses=0]
        %i47 = and i32 %i45, %i43               ; <i32> [#uses=1]
        %i48 = call avm2val @avm2box_i32( i32 %i47 )            ; <avm2val> [#uses=1]
        %i6 = call avm2val (...)* @avm2_bitand( avm2val %i4, avm2val %i44 )             ; <avm2val> [#uses=0]
        %i7 = call avm2val (...)* @avm2_convert_u( avm2val %i48 )               ; <avm2val> [#uses=3]
        %i63 = call i32 @avm2unbox_i32( avm2val %i7 )           ; <i32> [#uses=1]
        %i8 = call avm2val (...)* @avm2_pushuint( i32 0 )               ; <avm2val> [#uses=4]
        %i53 = call i32 @avm2unbox_i32( avm2val %i8 )           ; <i32> [#uses=1]
        %i49 = call i32 @avm2unbox_i32( avm2val %i8 )           ; <i32> [#uses=1]
        br label %bb_m6_b1_0_
        ; ...
```

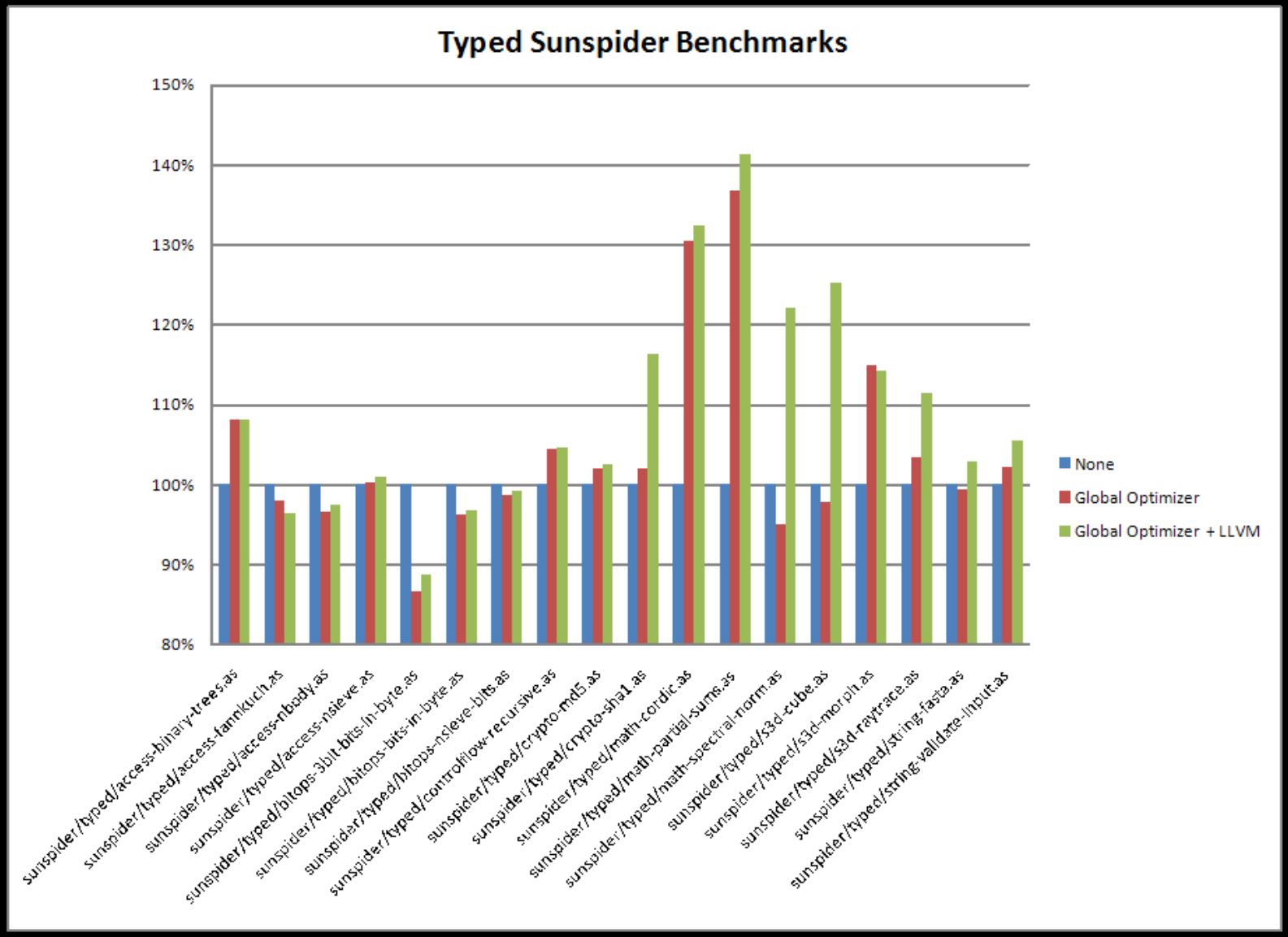
Results

- Mixed
  - Some meaningful positive results
  - Some substantial performance reductions

Results – higher numbers are better



Results – higher numbers are better



Where's my 20x speed increase? (or at least 2x!)

- Overhead avoided by Alchemy still dominates even well-optimized ABC
- Allocations
  - Up to nearly 50% of execution time (typed variant of Sunspider math-cordic)
  - Typed md5 – almost 30%
  - Typed nsieve-bits – over 30%
  - Typed FFT – over 30%
- AS3 Array access
  - Up to 75% of execution time (typed variant of Sunspider access-nsieve)
  - Typed fannkuch – almost 60%
- Dynamic property lookup
  - Up to nearly 45% of execution time (typed variant of Sunspider access-nbody)

Where's my 20x speed increase? (or at least 2x!)

- Value boxing
  - Typed fft– over 45%
  - Typed cordic – over 40%
  - Typed s3d-morph – over 40%
  - Typed md5 – over 30%
  - Typed sha1 – almost 30%
- VM's type inference is simple
  - Some optimizations change control flow such that a given value's type can no longer be deduced and becomes an expensive variant type
- Parameter passing in VM still expensive
  - Mitigated in some cases by inlining

Where's my 20x speed increase? (or at least 2x!)

- Still promising!

Futures

- Improve VM type deduction
- Continue refining GC
- Use LLVM to reduce some of the noted bottlenecks
  - Enable accurate GC to reduce mark load / enable object movement
  - Static escape analysis to reduce allocations
- Use LLVM analysis passes to enable AS3 specific optimizations
  - Type strengthening
  - Identify single-type Array usage
  - Identify Arrays with bounded sizes
  - Identify “prototype” OO uses that can convert to “real” classes (reducing dynamic lookups)
  - Identify explicit object deletion opportunities

Futures

- Extend tools to allow ahead of time, aggressively statically optimized compilation of AS3
  - Instead of generating calls to placeholder functions, call real functions in VM core
  - Could link against bitcode version of VM core, allowing AS3 to optimize against / inline pieces of existing C++ implementation
- Native versions of Flash/AIR libraries like Flex?
- Install time native codegen for AIR apps?
- Solution for platforms that don't allow JITs?


Adobe