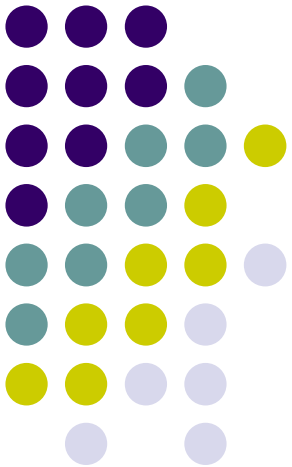
Execution Engine-independent JIT compilers

Michał Cierniak

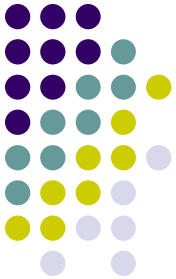
February 17, 2005



Low-level Intermediate Language (LIL)

- Problem: today's virtual machines are not well-structured and we are reaching limits of complexity that can be handled correctly
- Solution: LIL enables good abstractions without sacrificing performance
- This talk is not about refactoring interfaces but rather about separating policies from implementations

Benefits of LIL



- LIL allows better abstractions in the VEE design and gives “good enough” quality of the code generated by the JIT:
  - Adding a new EE implementation (e.g., of a method dispatch) becomes essentially free for the JIT
  - Inlining of features that don’t justify a lot of engineering becomes possible (e.g., multidimensional arrays)
  - Adding new features requires no or relatively few changes to JITs (e.g., traits)
- Analogous to using a high-level language vs. assembler

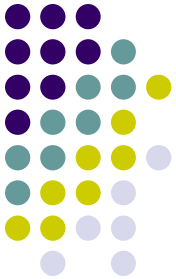
Outline

- Motivation
- LIL and progressive lowering
- Current implementation
- Related work
- Future work and conclusion

Open Runtime Platform

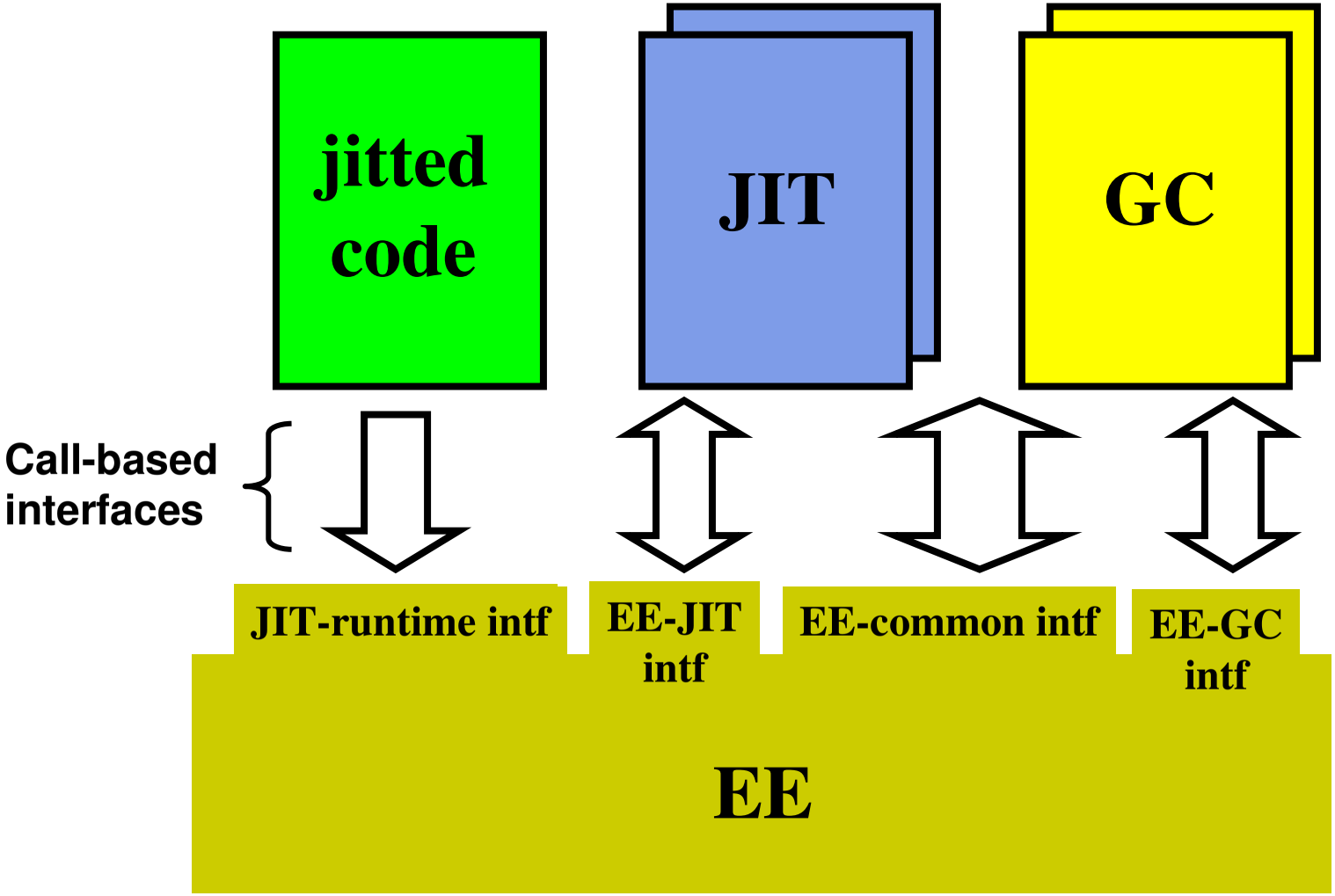
- Open Runtime Platform (ORP)
  - Support for ISO CLI (.NET CLR) and Java
  - Support for Itanium (IPF) and IA-32 (x86)
  - Runs on Windows and Linux
- ORP goals
  - Help hardware design
  - Software research
- Requirements
  - High performance (competitive with commercial products)
  - Flexible (e.g., study new HW/SW optimizations)
  - Robust (run real-world workloads)

Plug-in architecture

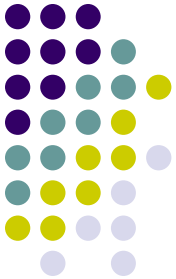


- ORP has a set of well-defined interfaces for JIT and GC modules
  - ORP uses 4 different JIT compilers
    - CLR also happens to have 4 different JITs
  - ORP uses 2 different garbage collectors
  - Since we started ORP in 1997 we had 7 JIT compilers and 5 garbage collectors
- Other interfaces (e.g., stack walking, exceptions) – not relevant to this talk

ORP interfaces

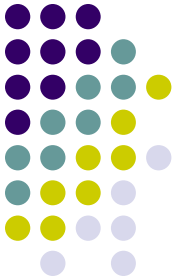


Why is plugging in a new JIT hard?



- We break abstractions for performance and convenience
  - Examples of hidden assumptions between components: object layout, method dispatch mechanism, when to initialize a class
- Lack of documentation
- Versioning of the JIT/EE interfaces

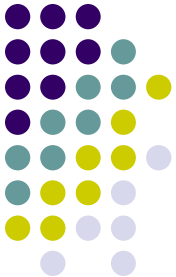
Considerations for a virtual call (a sample)



- Calling sequence, e.g. virtual method vs. interface method
- Binding, e.g. vtable slot offset for a virtual method
- Additional checks, e.g. CALLVIRT requires that a null check be done for the this argument (arg0).
- Hidden target parameters, e.g. type parameters for generic methods
- Allowed optimizations, e.g. can the call be inlined?
- Situational constraints, e.g. is profiling currently enabled?

Virtual method dispatch – an incomplete expansion

(pseudocode: lower case for compile-time ops, upper case may expand to actual instructions)



CALLVIRT:

```
if(method.isinterface)
  CALLINTF(method)
else if(method.isvirtual)
  CALLVIRTUAL(method)
else
  NULLCHECK(this)
  CALLNONVIRT(method)
```

Intentionally small font – do not attempt to read!

```
CALLVIRTUAL(token):
 if (token.hassecuritychecks)
  compiland.setforbidtailcalls
 if (compilerflags.profilecallreturn)
  DO_PROFILECALLHOOK(token, compiland)
  DO_BUILDVIRTUALCALLFRAME(token)
 if (token.isGenericcall)
  DO_GENERICVIRTUALCALL(token)
 elseif (token.isEnC)
  DO_ENCVIRTUALCALL(token)
 elseif (token.isfinal)
  DO_DIRECTVIRTUALCALL(token)
 else
  DO_NORMALVIRTUALCALL(token)
 if (token.hasRetBufArg)
  DO_COPYOUTARGS
 if (token.isvarargs)
  DO_DROPARGS
 elseif(compilerflags.callercleansstack)
  DO_CALLERCLEANCALLSTACK
 else
  DO_DROPCALLPAD
 DO_PUSHCALLRESULT(token)
 if (!token.hasRetBufArg)
  DO_NORMALIZEFPRETTYPE(token)
 if (compilerflags.profilecallreturn)
  DO_PROFILERETURNHOOK(compiland)
```

Inventing the arch

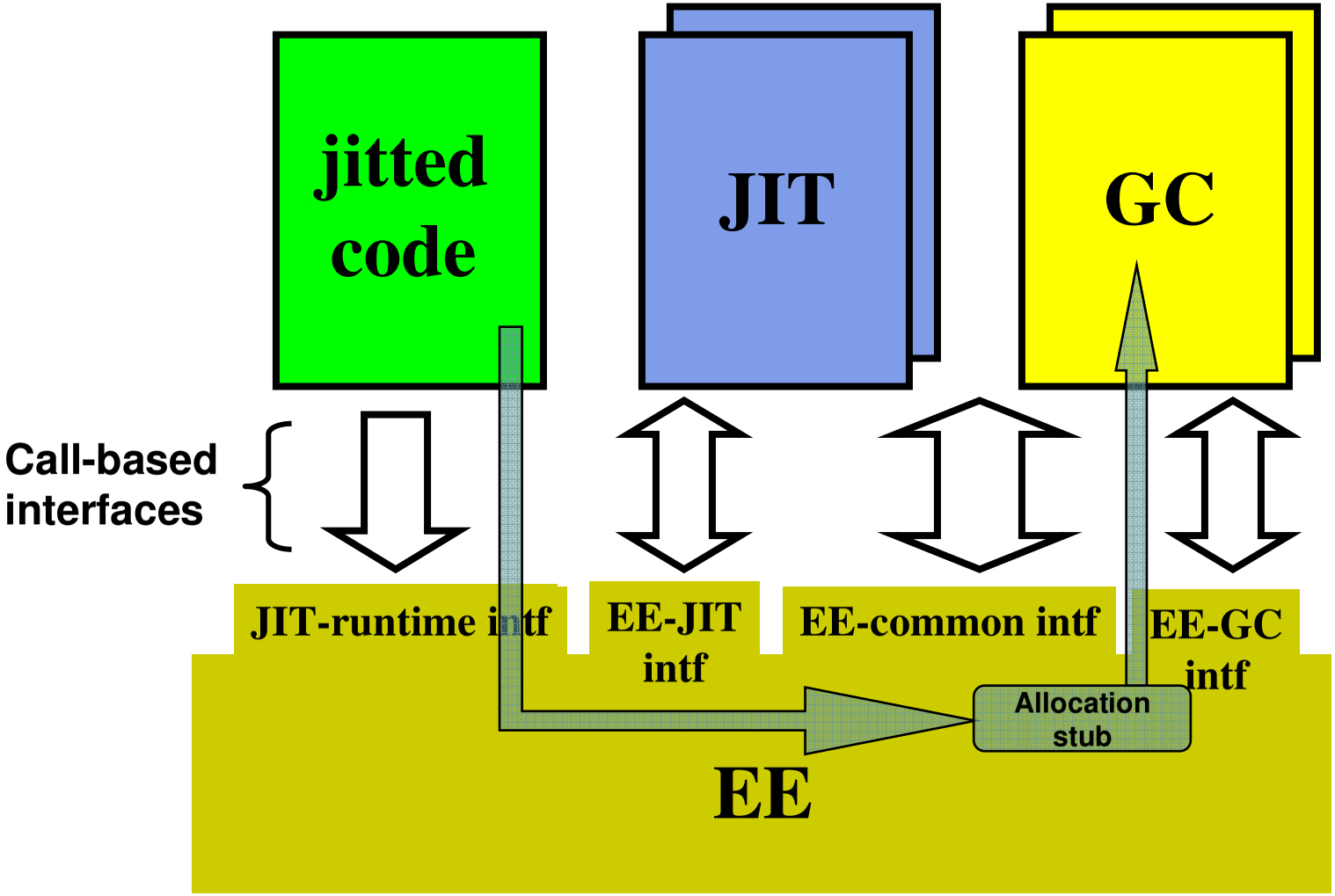
- Most software today is very much like an Egyptian pyramid with millions of bricks piled on top of each other, with no structural integrity, but just done by brute force and thousands of slaves.

Alan Kay

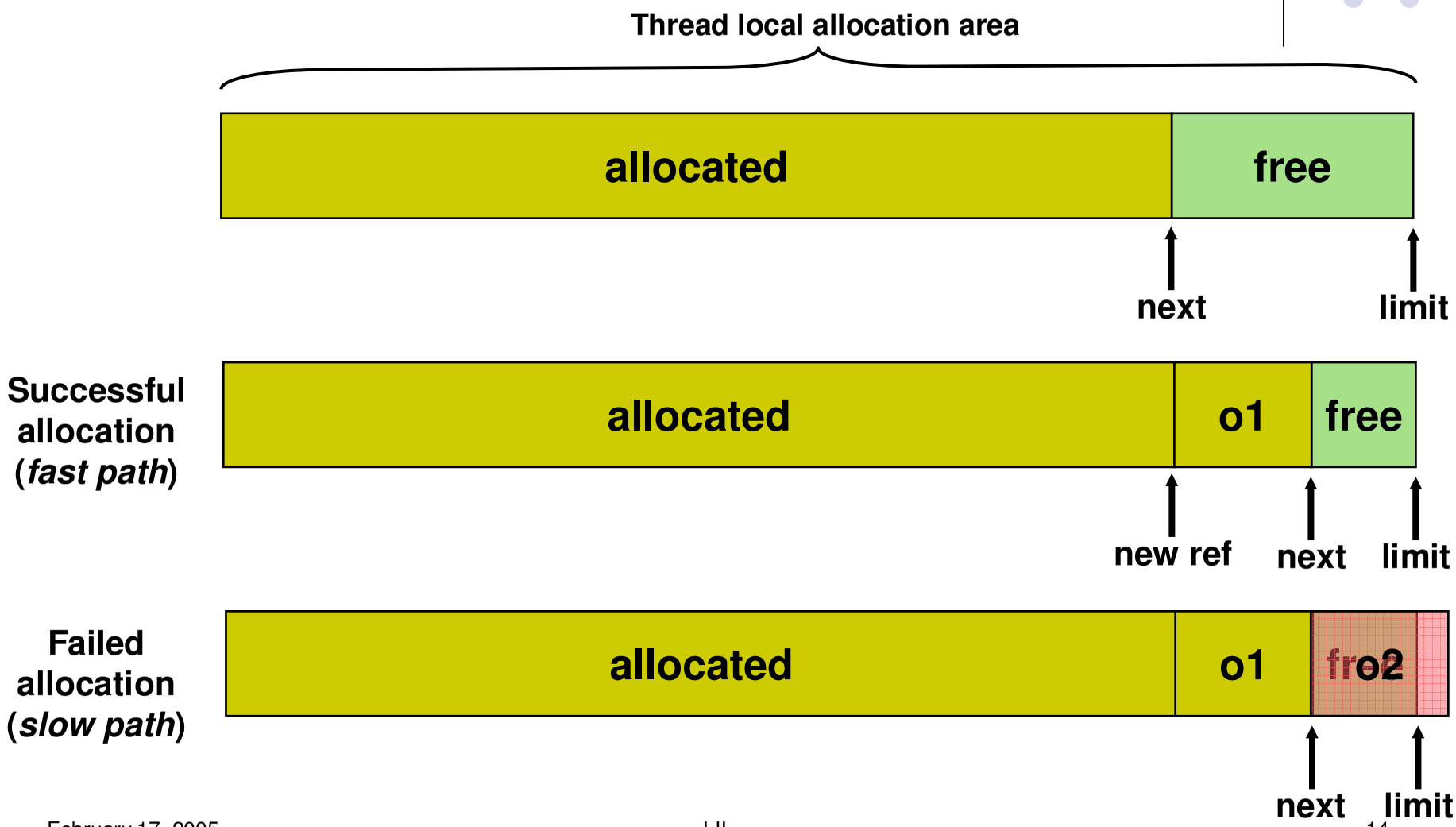
Outline

- Motivation
- **LIL and progressive lowering**
- Current implementation
- Related work
- Future work and conclusion

Object allocation in ORP



Object allocation



Object allocation without inlining

(fast path only)

Jitted code:

Inst 1
Inst 2
Inst 3
Inst 4
Call Alloc(sz, vt)
Inst 5
Inst 6
Inst 7
Inst 8

VM stub
(in LIL pseudocode):

```
Alloc(pint, pint): ref
o0 = i0
o1 = i1
o2 = tp + gcto
call GC_Alloc
br.null slow_path
ret
slow_path:
```

Legend:
tp: thread pointer
gcto: offset to GC data in thread local storage
i# - incoming arguments
o# - outgoing arguments

GC_Alloc (in C-like pseudocode):

```
Byte *
GC_Alloc(int sz,
         VT vt,
         GC_TLS *gc_tls)
// vt = Vtable pointer
// sz = object size
// gc_tls = thread local GC info
{
  Byte *next = gc_tls->next;
  if(gc_tls->limit - next >= sz) {
     gc_tls->next = next + sz;
     *(VT *)next = vt;
     return next;
  } else {
     return NULL
  }
}
```

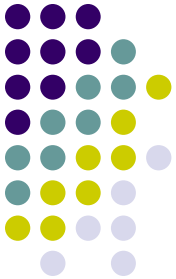
Hand-tuned allocation sequence

(in IPF-like pseudocode)

tp – Base for TLS (Thread-Local Storage)

```
        g1 = r4 + offset_gc_next
        g2 = r4 + offset_gc_limit
        r8 = [g1]       // r8 = next
        g3 = [g2]       // g3 = limit
        g4 = r8 + i0    // g4 = next + size
        p1 = (g4 <= g3)
(p1)  [g1] = g4         // next = g4
(p1)  [r8] = i1         // store vtable ptr
(p1)  return            // fast path succeeded
      // fall through to the slow path
```

TLS instruction sequence



Win IPF

```
g1 = r4 + offset_gc_next
```

Win IA-32

```
eax = [fs:0x14]
eax = eax + offset_gc_next
```

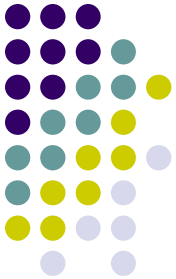
Linux IA-32

```
push &TLS_key_porpthread
call pthread_getspecific
add esp,4
eax = eax + offset_gc_next
```

Xscale…
WinCE…

```
???
???
```

Why not have the JIT inline the VM functionality?



- ORP is evolving. During 2002 and 2003:
  - tp materialization changed multiple times
  - allocation sequence changed multiple times
  - Even today there are three tp materializations (Linux IA-32, Win32, and Win64)!
- Each change would require a modification of the JIT
- Allocation and tp materialization are just examples
- Other examples: method dispatch, synchronization, type checks, …
- This approach does not scale!

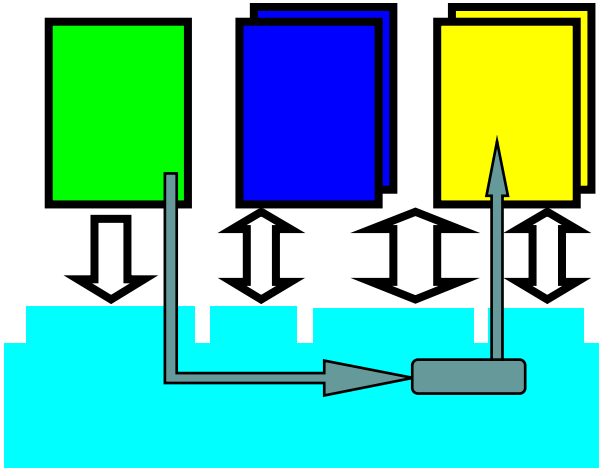
Solution: LIL

- An implementation-independent language for code implants:

1. GC implements Alloc in LIL
2. EE expands TLS in the Alloc sequence
3. JIT inlines Alloc into the jitted code

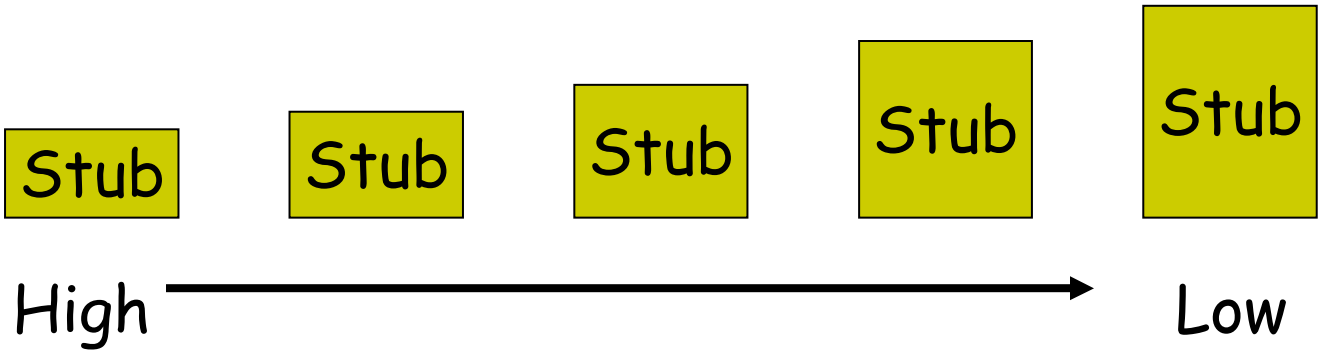
Problems solved

What was the original motivation for two calls needed to allocate an object?



- The GC should not know how thread local storage (TLS) is implemented
  - Need a high-level abstraction of TLS
- The JIT should know neither the TLS nor GC implementations
  - Need a low-level expansion of TLS operations
- LIL becomes two languages:
  - High-LIL – high-level, with EE primitives
  - Low-LIL – low-level, close to CPU instructions

Progressive lowering



- This idea of lowering should be applied to multiple modules in the EE
- But we need to strike the right balance
  - Address phase ordering
  - Make sure that the distributed policy still be comprehensible by developers

Alloc in LIL

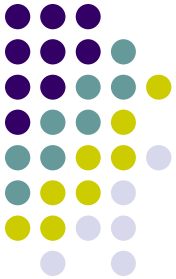
```
entry 0:managed:pint,pint:ref;
locals 3;
L0 = ts;
ld r, [L0 + next_offs: pint];
ld L1, [L0 + limit_offs: pint];
L2 = r: pint + i0: pint;
jc l2 > L1, slowpath;
st [r: + 0 :pint], i1;
st [L0 + next_offs: pint], L2;
ret;
```

```
:slowpath;
push_m2n 0;
in2out platform:ref;
call gc_alloc;
pop_m2n;
ret;
```

Outline

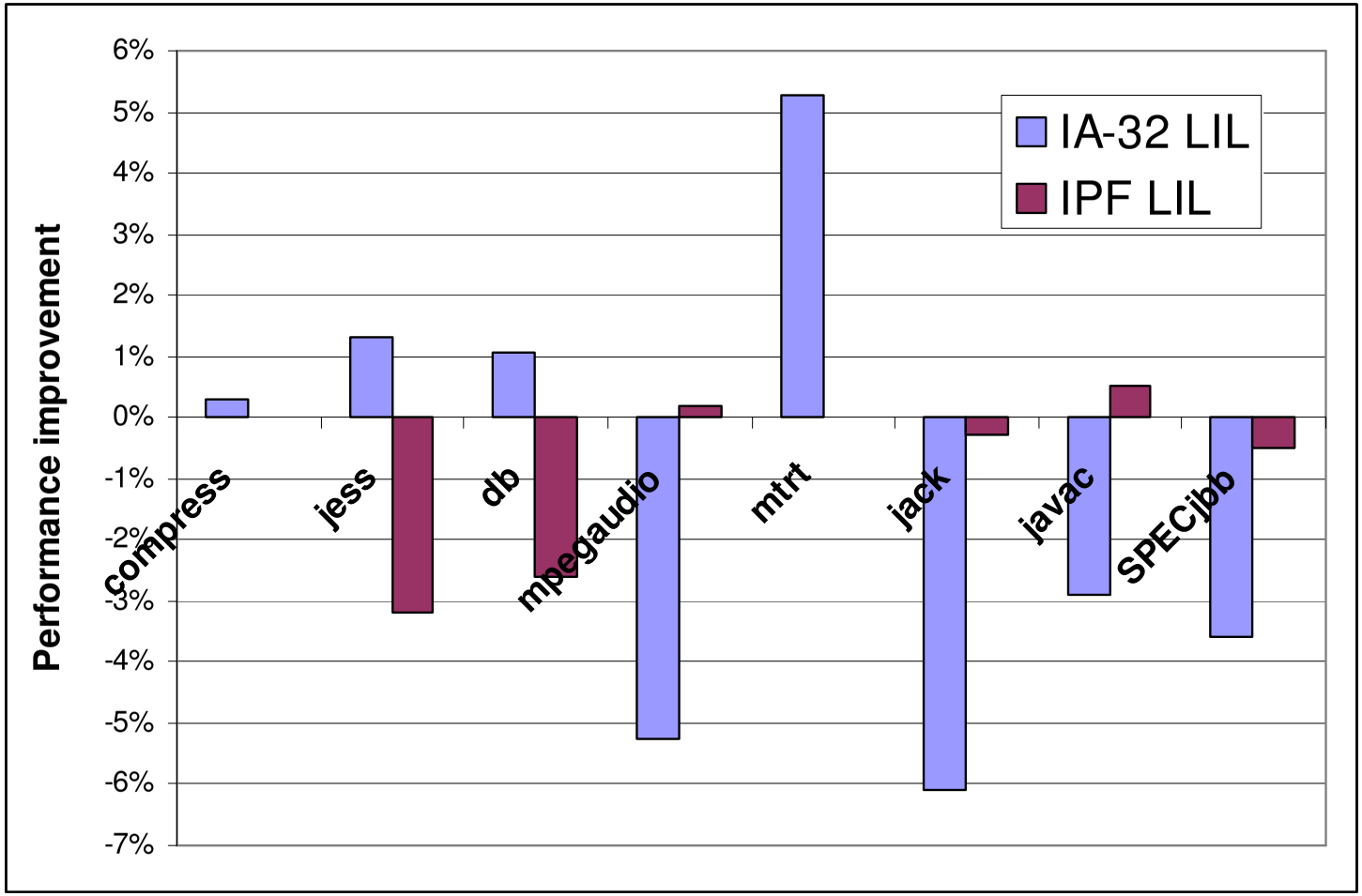
- Motivation
- LIL and progressive lowering
- **Current implementation**
- Related work
- Future work and conclusion

Implemented LIL stubs (IA-32 and IPF)



- Compilation: compile-me generic, compile-me specific, recompile
- Native-method interface: JNI and PInvoke stubs
- Exceptions: throw, lazy throw, throw specific exceptions, throw linking exception
- Allocation: new object, new array, multinewarray, load constant string
- Synchronization: monitor enter and exit for objects and classes
- Type tests: checkcast, instanceof, aastore
- Arithmetic helpers: float to integer, double to integer, long shifts, long multiplies, long divides
- Miscellaneous: load interface vtable, initialize class, character-array copy

Performance of LIL stubs



“Littleness” of LIL

(LOC = Lines Of Code)

- 3,500 LOC – parser, infrastructure
- 1,500 LOC – IA-32 code generator
- 2,200 LOC – IPF code generator

Type checks

- Slow version:
  - ORP provides a function that can be called at run time to implement the checkcast instruction

```
push class_id
push eax  // object reference
call checkcast
```

- Fast version
  - *Display* (constant time for common cases)

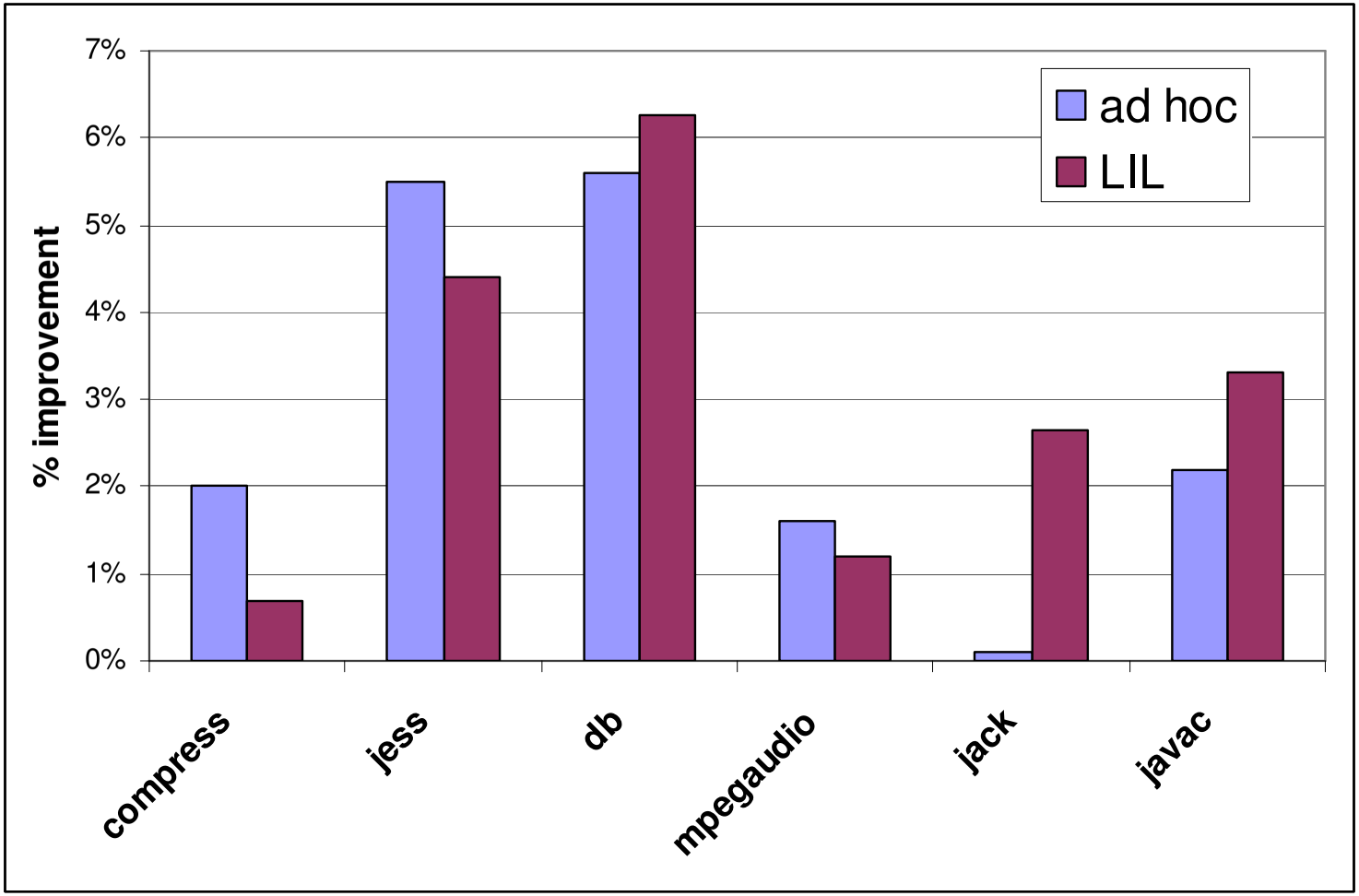
```
mov  eax, [eax]   // get vtable ptr
mov  eax, [eax + offs + (depth - 1) * 4]
cmp  eax, class_id
```

Checkcast in LIL

- Type test (JVM checkcast instruction) for a given class c:

```
entry 0: managed: ref: void;
jc i0 != 0, nonnull;
ret;
:nonnull;
locals 1;
ld L0,[i0 + vtable_offset: pint];
ld L0,[L0 + ato + 4 * (depth - 1): pint];
jc L0 != c, failed;
ret;
:failed;
out platform::void;
call.noret throw_class_cast_exception;
```

Performance of inlining LIL into IA-32 JIT



Outline

- Motivation
- LIL and progressive lowering
- Current implementation
- **Related work**
- Future work and conclusion

Related work

- Using managed code
  - Bartok (Singularity)
  - JikesRVM
- Low-level common abstraction
  - SEAM/Alice
  - C--
  - LLVM (Low Level Virtual Machine)
  - PCR (Portable Common Runtime)

Comparison of related work

	CLR	PCR	LLVM	C--	SEAM	Bartok	**LIL**
High-Level IL	Y	N	N	N	N	Y	**Y**
Fully language-independent	N	Y	Y	Y	Y	N	**1/2**
JIT independent from EE	N	N	Y	Y	Y	1/2	**Y**
Compiler must generate RT	N	1/2	Y	Y	Y	N	**N**
Compiler must generate JIT	N	N/A	N	N	Y	N	**N**

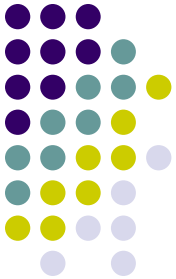
Outline

- Motivation
- LIL and progressive lowering
- Current implementation
- Related work
- **Future work and conclusion**

Future work

- Expressing constraints
- More functions inlined by the JIT
- Comprehensive analysis of the performance gap between a JIT that breaks the EE abstractions and one that uses LIL
- In the CLR, consider designing an MSIL-based language with LIL properties

Expressing constraints



- Consider expansion of the stfld instruction:

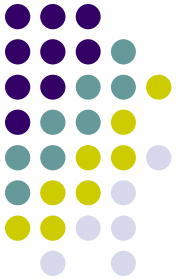
```
STFLD(fieldtoken) :
    ENSURE_CLASSINIT(fieldtoken.classhandle)
    DO_WRITE_BARRIER(fieldtoken)
    DO_STFLD(fieldtoken)
```

  - GC not allowed between write barrier and the write to the field
- When a class should be initialized?

More information about LIL

- N. Glew, S. Triantafyllis, M. Cierniak, M. Eng, B. Lewis and J. Stichnoth. *LIL: An Architecture-Neutral Language for Virtual-Machine Stubs*. VM'04, May 2004.
- M. Cierniak, N. Glew, S. Triantafyllis, M. Eng, B. Lewis and J. Stichnoth. *Object-Model Independence via Code Implants*. Workshop on Multiparadigm Programming with OO Languages (MPOOL'03), October 2003

Summary



- Hidden assumptions between VEE components are very hard to manage
- LIL makes this problem more manageable
  - Easier to understand what code to generate
  - Progressive lowering is the key insight
- Benefits
  - Higher confidence that generated code is correct
  - Easier to write a new JIT (or GC, etc)
  - Easier to change EE features