LLVM - the early days

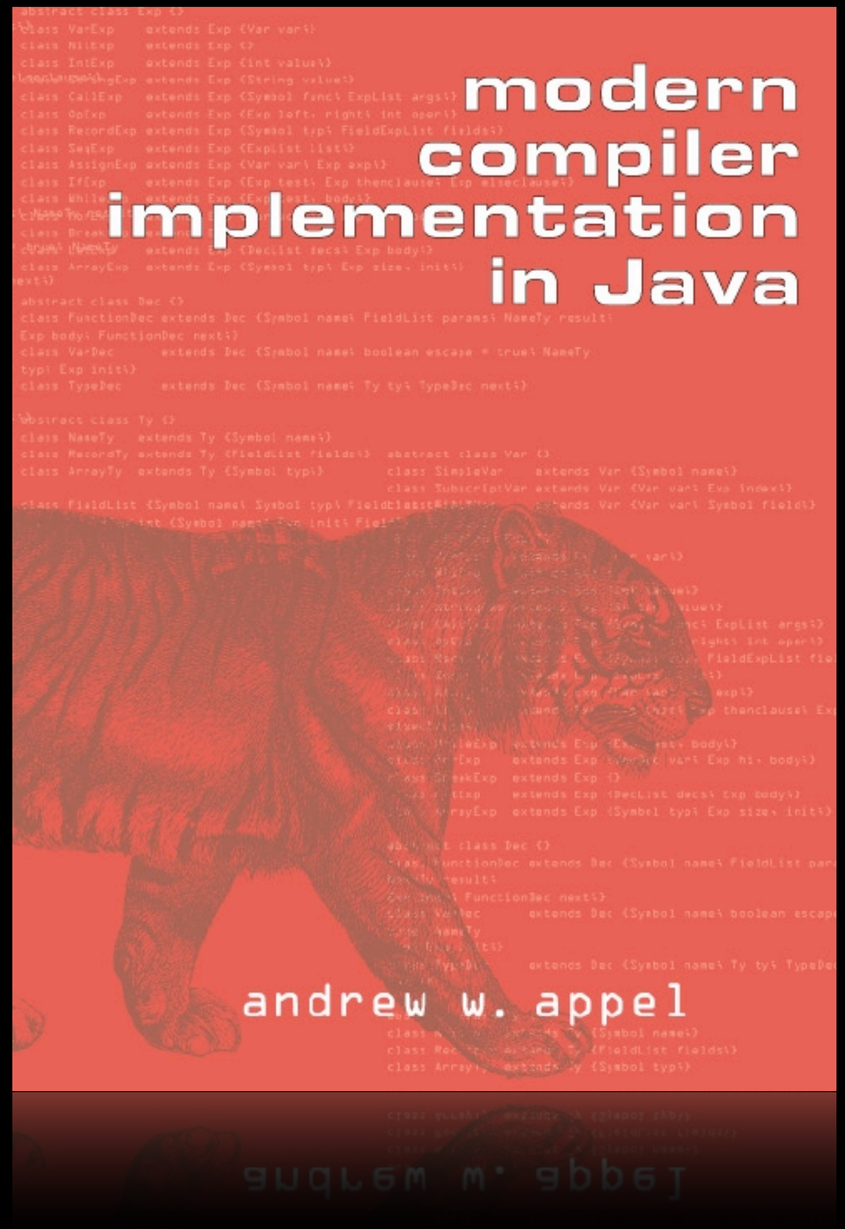
Where did it come from, and how?

Before LLVM

September 1999

Tiger native compiler

- Directed study in compilers @ UofP:
  - with Dr. Steven Vegdahl, Nick Forrette

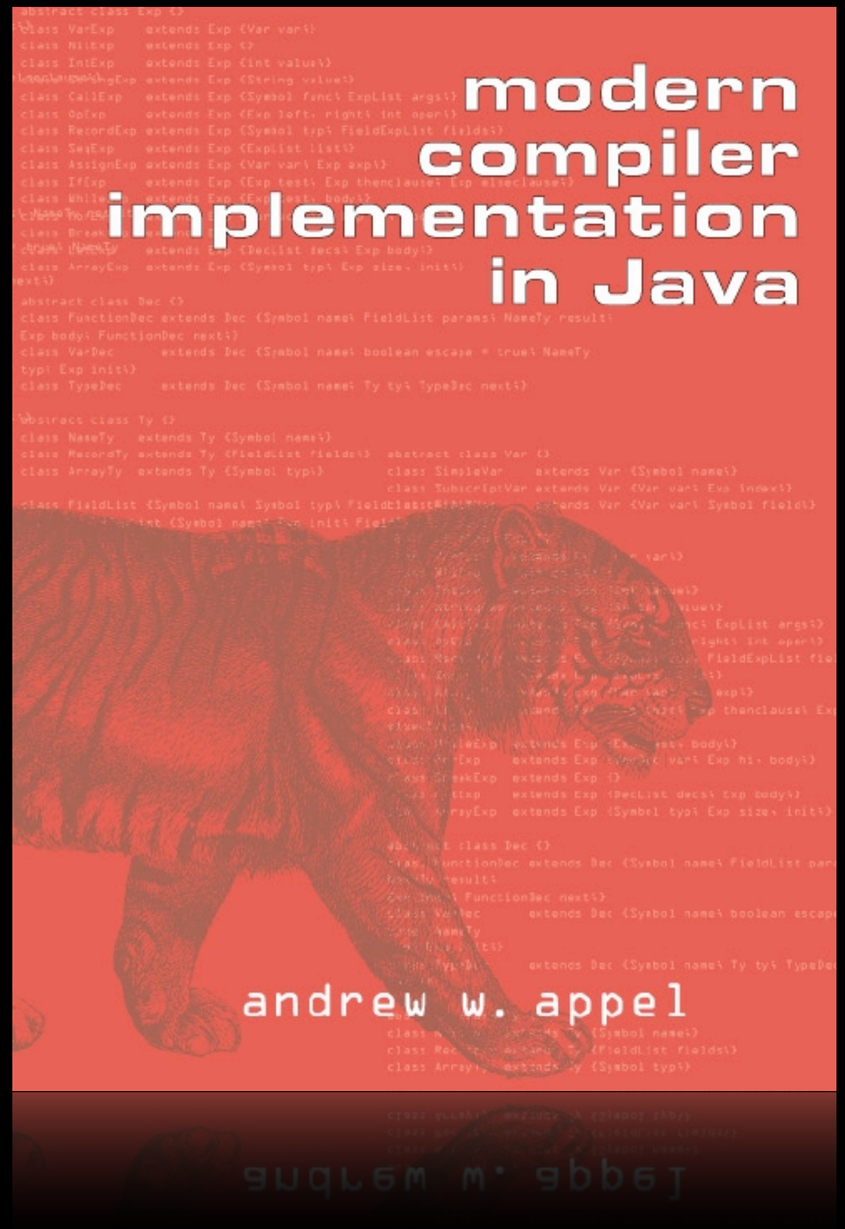


http://www.nondot.org/sabre/Projects/Compilers/

Tiger native compiler

- Directed study in compilers @ UofP:
  - with Dr. Steven Vegdahl, Nick Forrette

- Built a full native compiler in Java:
  - “Tiger” to X86 assembly

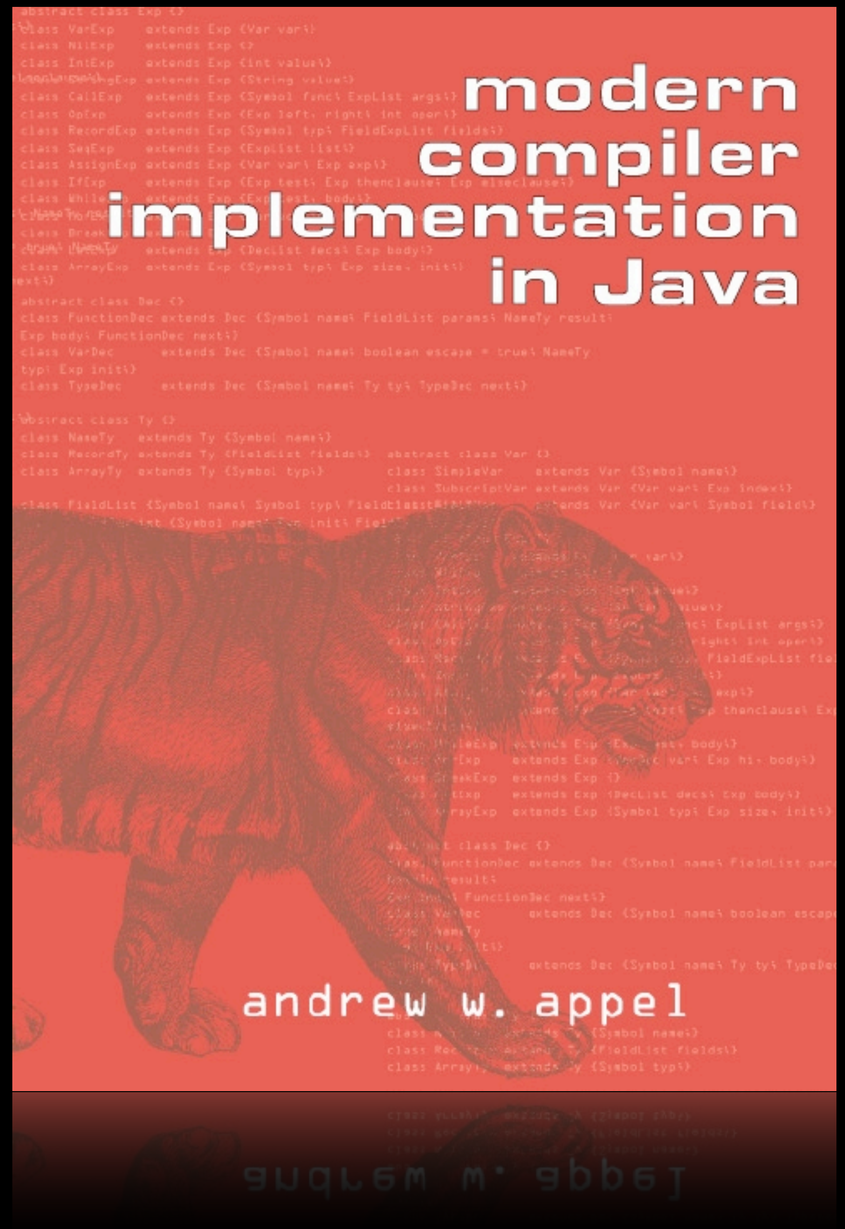


http://www.nondot.org/sabre/Projects/Compilers/

Tiger native compiler

- Directed study in compilers @ UofP:
  - with Dr. Steven Vegdahl, Nick Forrette

- Built a full native compiler in Java:
  - “Tiger” to X86 assembly
- Full runtime:
  - Written in C and Assembly
  - Included a copying GC with accurate stack scanning



http://www.nondot.org/sabre/Projects/Compilers/

IRWIN

"Intermediate Representation With Interesting Name"

```
sub printInt(X)
  local T
  local D

  T = X < 0xA
  D = X >= 0
  T = T & D     ; if X >= 0 && X < 10.
  if T goto PrintIntHelperBaseCase

  ; Multidigit number.
  T = X / 0xA    ; High order digits
  D = X % 0xA    ; Low order digit.
  call _printIntHelper(T)
  X = D

PrintIntHelperBaseCase:
  X = X + '0'
  call __putch(X)
  return
end sub _printIntHelper
```

IRWIN

"Intermediate Representation With Interesting Name"

```
sub printInt(X)
  local T
  local D

  T = X < 0xA
  D = X >= 0
  T = T & D      ; if X >= 0 && X < 10.
  if T goto PrintIntHelperBaseCase

  ; Multidigit number.
  T = X / 0xA    ; High order digits
  D = X % 0xA    ; Low order digit.
  call _printIntHelper(T)
  X = D

PrintIntHelperBaseCase:
  X = X + '0'
  call __putch(X)
  return
end sub _printIntHelper
```

- First class textual format

IRWIN

"Intermediate Representation With Interesting Name"

```
sub printInt(X)
  local T
  local D

  T = X < 0xA
  D = X >= 0
  T = T & D       ; if X >= 0 && X < 10.
  if T goto PrintIntHelperBaseCase

  ; Multidigit number.
  T = X / 0xA     ; High order digits
  D = X % 0xA     ; Low order digit.
  call _printIntHelper(T)
  X = D

PrintIntHelperBaseCase:
  X = X + '0'
  call __putch(X)
  return
end sub _printIntHelper
```

- First class textual format
- Three address code

IRWIN

"Intermediate Representation With Interesting Name"

```
sub printInt(X)
  local T
  local D

  T = X < 0xA
  D = X >= 0
  T = T & D      ; if X >= 0 && X < 10.
  if T goto PrintIntHelperBaseCase

  ; Multidigit number.
  T = X / 0xA    ; High order digits
  D = X % 0xA    ; Low order digit.
  call _printIntHelper(T)
  X = D

PrintIntHelperBaseCase:
  X = X + '0'
  call __putch(X)
  return
end sub _printIntHelper
```

- First class textual format
- Three address code
- Unlimited register file - not SSA

IRWIN

"Intermediate Representation With Interesting Name"

```
sub printInt(X)
  local T
  local D

  T = X < 0xA
  D = X >= 0
  T = T & D      ; if X >= 0 && X < 10.
  if T goto PrintIntHelperBaseCase

  ; Multidigit number.
  T = X / 0xA    ; High order digits
  D = X % 0xA    ; Low order digit.
  call _printIntHelper(T)
  X = D

PrintIntHelperBaseCase:
  X = X + '0'
  call __putch(X)
  return
end sub _printIntHelper
```

- First class textual format
- Three address code
- Unlimited register file - not SSA
- Functions

IRWIN

"Intermediate Representation With Interesting Name"

```
sub printInt(X)
  local T
  local D

  T = X < 0xA
  D = X >= 0
  T = T & D      ; if X >= 0 && X < 10.
  if T goto PrintIntHelperBaseCase

  ; Multidigit number.
  T = X / 0xA    ; High order digits
  D = X % 0xA    ; Low order digit.
  call _printIntHelper(T)
  X = D

PrintIntHelperBaseCase:
  X = X + '0'
  call __putch(X)
  return
end sub _printIntHelper
```

- First class textual format
- Three address code
- Unlimited register file - not SSA
- Functions
- Control flow

IRWIN

"Intermediate Representation With Interesting Name"

```
sub printInt(X)
  local T
  local D

  T = X < 0xA
  D = X >= 0
  T = T & D      ; if X >= 0 && X < 10.
  if T goto PrintIntHelperBaseCase

  ; Multidigit number.
  T = X / 0xA    ; High order digits
  D = X % 0xA    ; Low order digit.
  call _printIntHelper(T)
  X = D

PrintIntHelperBaseCase:
  X = X + '0'
  call __putch(X)
  return
end sub _printIntHelper
```

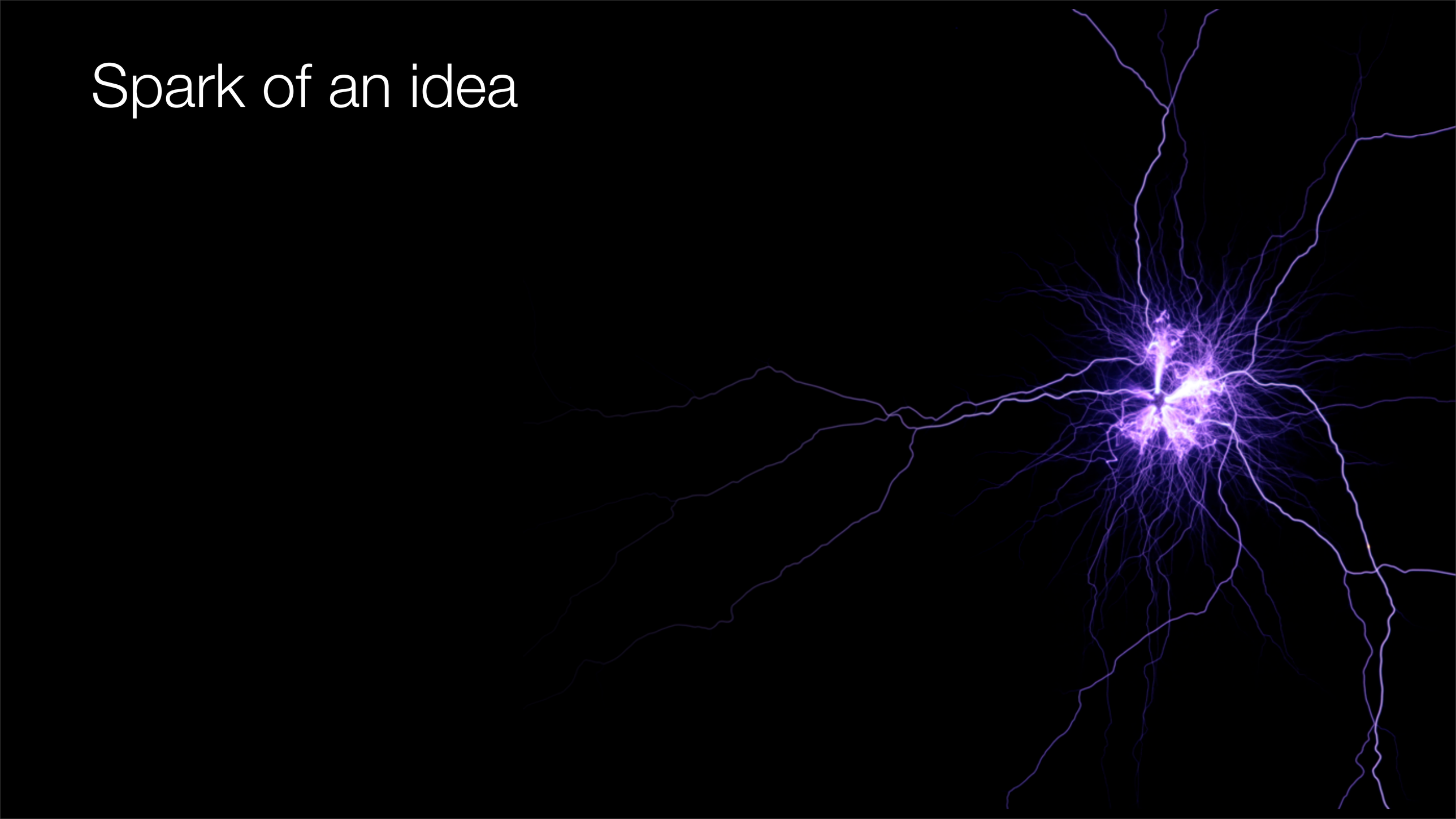
- First class textual format
- Three address code
- Unlimited register file - not SSA
- Functions
- Control flow

- No type system
- Syntactic travesty

Conception

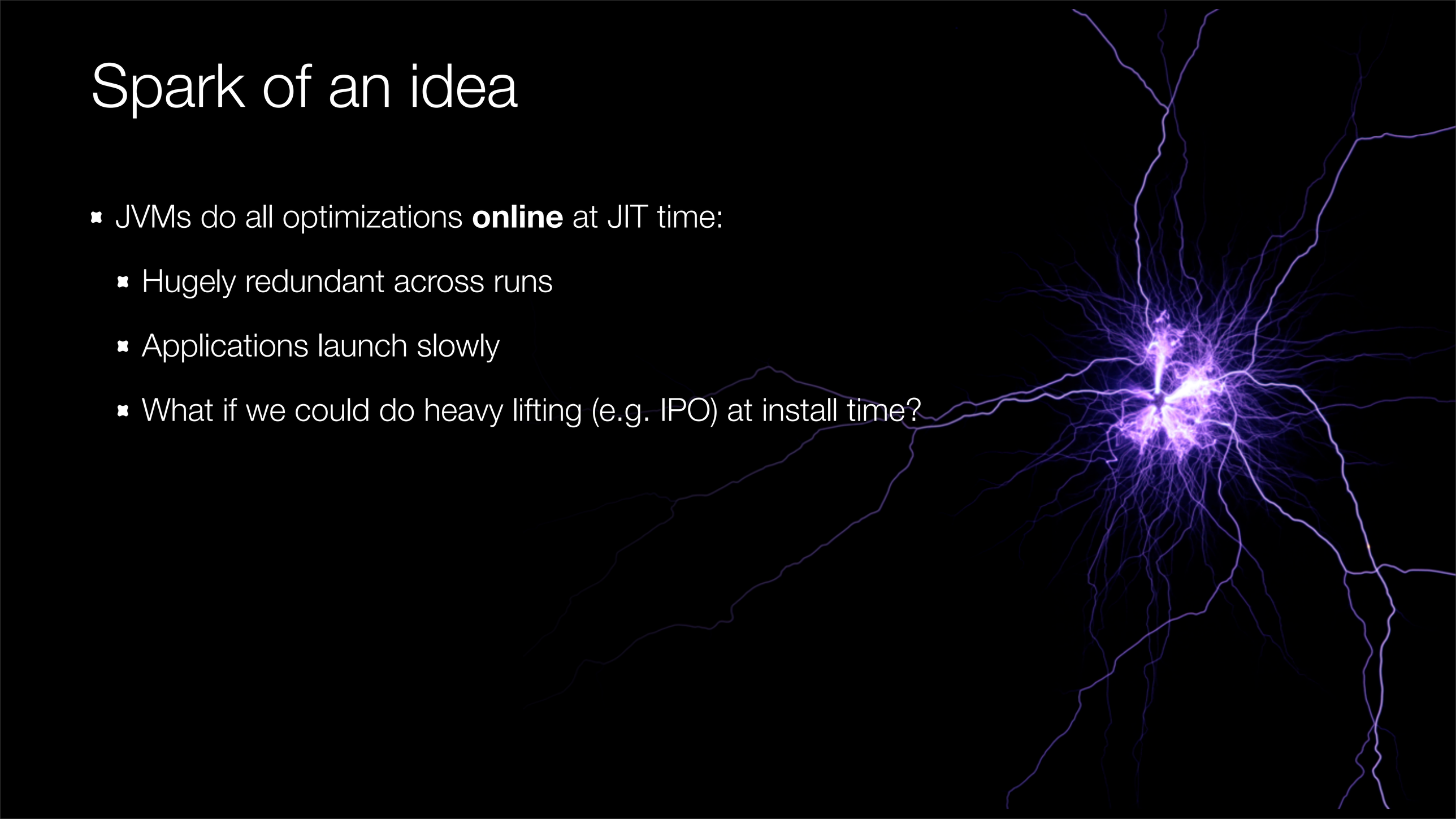
December 2000

Spark of an idea



Spark of an idea

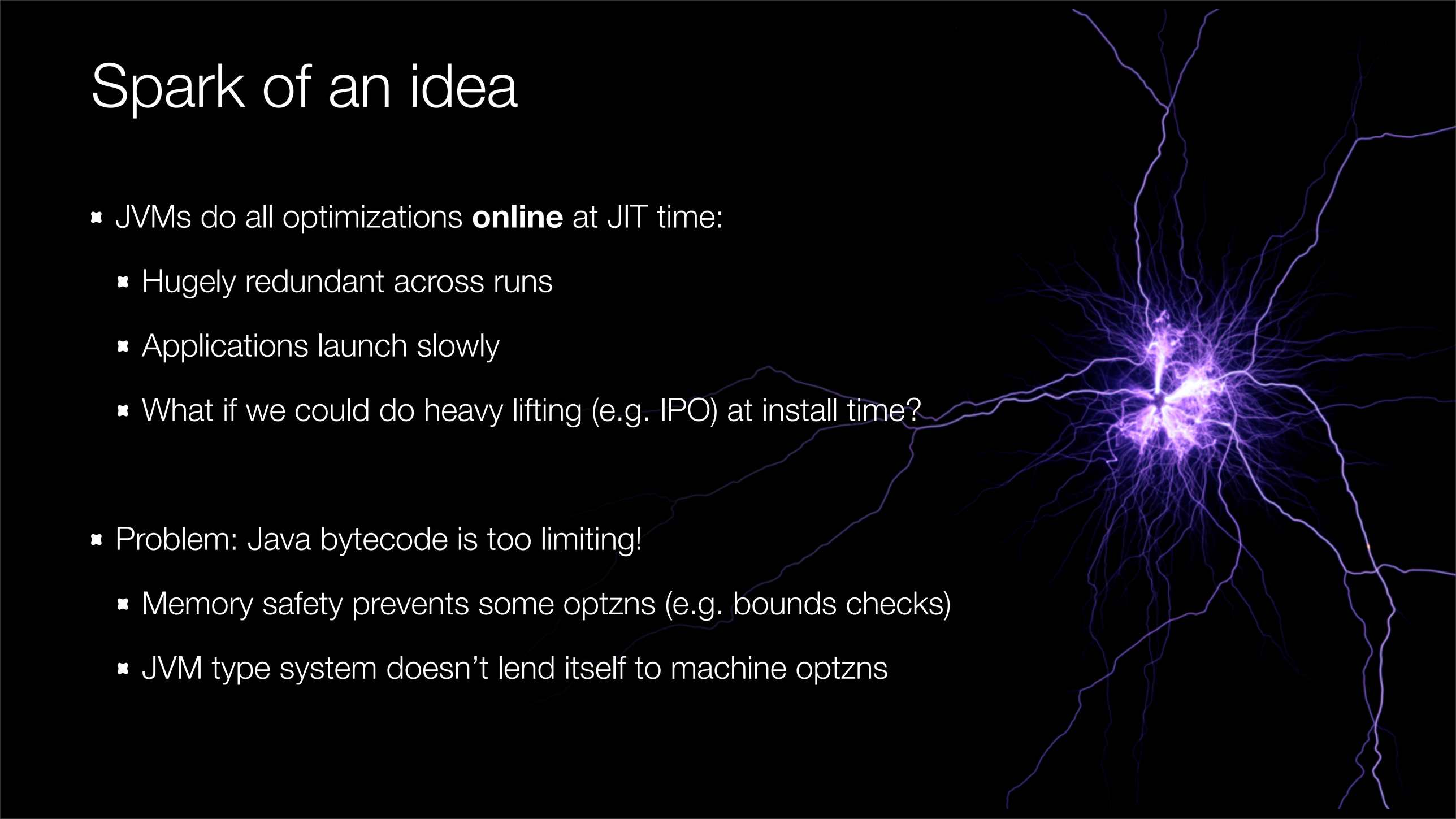
- JVMs do all optimizations **online** at JIT time:
  - Hugely redundant across runs
  - Applications launch slowly
  - What if we could do heavy lifting (e.g. IPO) at install time?



Spark of an idea

- JVMs do all optimizations **online** at JIT time:
  - Hugely redundant across runs
  - Applications launch slowly
  - What if we could do heavy lifting (e.g. IPO) at install time?

- Problem: Java bytecode is too limiting!
  - Memory safety prevents some optzns (e.g. bounds checks)
  - JVM type system doesn’t lend itself to machine optzns



“With some sort of low level virtual machine, we could optimize better and a JIT compiler would have to do less work online!”

Winter Break

January 2001

First prototype of LLVM

- 9676 lines of C++ code

http://nondot.org/sabre/llvm-one-month-old.tar.gz

First prototype of LLVM

- 9676 lines of C++ code
- as, dis, opt
  - Textual IR and bytecode

http://nondot.org/sabre/llvm-one-month-old.tar.gz

First prototype of LLVM

- 9676 lines of C++ code
- as, dis, opt
  - Textual IR and bytecode
- Two simple optimizations
  - Constant Propagation
  - Dead Code elimination

http://nondot.org/sabre/llvm-one-month-old.tar.gz

Familiar Structure

```
llvm/
      include/llvm/


      lib/


      tools/
```

Familiar Structure

```
llvm/
      include/llvm/


      lib/









      tools/
            as/
            dis/
            opt/
```

Familiar Structure

```
llvm/
     include/llvm/
                  Assembly/

     lib/
         VMCore/
         Assembly/{Parser/, Writer/}
         Bytecode/{Reader/, Writer/}
         MethodAnalysis/
         Optimizations/

     tools/
           as/
           dis/
           opt/
```

Familiar Structure

```
llvm/
     include/llvm/
                  Assembly/

     lib/
         VMCore/                                  "IR" in 2013
         Assembly/{Parser/, Writer/}
         Bytecode/{Reader/, Writer/}              "Bitcode" in LLVM 2.0
         MethodAnalysis/                          "Analysis" in 2001
         Optimizations/                           "Transforms" in 2001

     tools/
           as/                                    llvm-as in 2001
           dis/                                   llvm-dis in 2001
           opt/
```

include/llvm

include/llvm

BasicBlock.h
Class.h
Def.h
DerivedTypes.h
InstrTypes.h
Instruction.h
Instructions.h
Method.h
SymTabValue.h
SymbolTable.h
Type.h
Value.h
ValueHolder.h
ValueHolderImpl.h

Header Style

```
//===-- llvm/DerivedTypes.h - Classes for handling data types ----*- C++ -*-=//
//
// This file contains the declarations of classes that represent "derived
// types".  These are things like "arrays of x" or "structure of x, y, z" or
// "method returning x taking (y,z) as parameters", etc...
//
// The implementations of these classes live in the Type.cpp file.
//
//===----------------------------------------------------------------------===//

#ifndef LLVM_DERIVED_TYPES_H
#define LLVM_DERIVED_TYPES_H

#include "llvm/Type.h"
```

Header Style

```
//===-- llvm/DerivedTypes.h - Classes for handling data types ----*- C++ -*-=//
//
// This file contains the declarations of classes that represent "derived
// types".  These are things like "arrays of x" or "structure of x, y, z" or
// "method returning x taking (y,z) as parameters", etc...
//
// The implementations of these classes live in the Type.cpp file.
//
//===----------------------------------------------------------------------===//

#ifndef LLVM_DERIVED_TYPES_H
#define LLVM_DERIVED_TYPES_H

#include "llvm/Type.h"



// Future derived types: pointer, array, sized array, struct, SIMD packed format
```

llvm/Makefile.common

```
#                                     Makefile.common
#
# This file is included by all of the LLVM makefiles.  This file defines common
# rules to do things like compile a .cpp file or generate dependancy info.
# These are platform dependant, so this is the file used to specify these
# system dependant operations.
#
# The following functionality may be set by setting incoming variables:
#
# 1. LEVEL - The level of the current subdirectory from the top of the
#    MagicStats view.  This level should be expressed as a path, for
#    example, ../.. for two levels deep.
#
# 2. DIRS - A list of subdirectories to be built.  Fake targets are set up
#    so that each of the targets "all", "install", and "clean" each build.
#    the subdirectories before the local target.
#
# 3. Source - If specified, this sets the source code filenames.  If this
#    is not set, it defaults to be all of the .cpp, .c, .y, and .l files
#    in the current directory.
#
```

Value.h - RAUW!

```
class Value {
public:
..



// replaceAllUsesWith - Go through the uses list for this definition and make
// each use point to "D" instead of "this".  After this completes, 'this's
// use list should be empty.
//
void replaceAllUsesWith(Value *D);

//----------------------------------------------------------------------
// Methods for handling the list of uses of this DEF.
//
typedef list<Instruction*>::iterator       use_iterator;
typedef list<Instruction*>::const_iterator use_const_iterator;

inline bool              use_size()  const { return Uses.size();  }
inline use_iterator      use_begin()       { return Uses.begin(); }
inline use_const_iterator use_begin() const { return Uses.begin(); }
inline use_iterator      use_end()         { return Uses.end();   }
inline use_const_iterator use_end()  const { return Uses.end();   }
```

Partial Class Hierarchy

```
Value
    Def                                  Became “User”
        MethodArgument                   Became “Argument”
        Instruction
```

Partial Class Hierarchy

```
Value
    Def                                          Became “User”
        MethodArgument                           Became “Argument”
        Instruction
            PHINode
            CallInst
            UnaryOperator
            BinaryOperator
```

Partial Class Hierarchy

```
Value
    Def                                          Became “User”
        MethodArgument                           Became “Argument”
        Instruction
            PHINode
            CallInst
            UnaryOperator
            BinaryOperator
            TerminatorInst
              ReturnInst
              BranchInst
              SwitchInst
```

LLVM IR Syntax

```
class "TestClass" {

  int "func"(int, int)
    int 0
  {
;  int func(int %i0, int %j0) {
;   %i1 = add int %i0, $0   ; Names are started by %, constants $
    add int -1, -2          ; => 3
    add int -1, -3          ; => 4
    setle int -1, 0         ; => bool 0
    br 0, 1, 2

;  BB1:
    add int -1, -4          ; => 5
    br 3                    ;    br BB3

;  BB2:
    sub int -2, -5          ; => 6
    br 3                    ;    br BB3

;  BB3:
    phi int -1, 0           ; => 7
    add int -3, -7          ; => 8
    add int -1, 0           ; => 9
    ret int 0
  }
}
```

LLVM IR Syntax

```
class "TestClass" {

  int "func"(int, int)
    int 0
  {
;  int func(int %i0, int %j0) {
;    %i1 = add int %i0, $0   ; Names are started by %, constants $
    add int -1, -2          ; => 3
    add int -1, -3          ; => 4
    setle int -1, 0         ; => bool 0
    br 0, 1, 2

;  BB1:
    add int -1, -4          ; => 5
    br 3                    ;    br BB3

;  BB2:
    sub int -2, -5          ; => 6
    br 3                    ;    br BB3

;  BB3:
    phi int -1, 0           ; => 7
    add int -3, -7          ; => 8
    add int -1, 0           ; => 9
    ret int 0
  }
}
```

- Familiar opcodes
- LLVM 1.0 C-style type system

LLVM IR Syntax

```
class "TestClass" {

  int "func"(int, int)
    int 0
  {
;  int func(int %i0, int %j0) {
;   %i1 = add int %i0, $0   ; Names are started by %, constants $
    add int -1, -2          ; => 3
    add int -1, -3          ; => 4
    setle int -1, 0         ; => bool 0
    br 0, 1, 2

;  BB1:
    add int -1, -4          ; => 5
    br 3                    ;    br BB3

;  BB2:
    sub int -2, -5          ; => 6
    br 3                    ;    br BB3

;  BB3:
    phi int -1, 0           ; => 7
    add int -3, -7          ; => 8
    add int -1, 0           ; => 9
    ret int 0
  }
}
```

- Familiar opcodes
- LLVM 1.0 C-style type system
- General syntax direction understood

LLVM IR Syntax

```
class "TestClass" {

  int "func"(int, int)
    int 0
  {
;   int func(int %i0, int %j0) {
;    %i1 = add int %i0, $0    ; Names are started by %, constants $
     add int -1, -2           ; => 3
     add int -1, -3           ; => 4
     setle int -1, 0          ; => bool 0
     br 0, 1, 2

;  BB1:
     add int -1, -4           ; => 5
     br 3                     ;    br BB3

;  BB2:
     sub int -2, -5           ; => 6
     br 3                     ;    br BB3

;  BB3:
     phi int -1, 0            ; => 7
     add int -3, -7           ; => 8
     add int -1, 0            ; => 9
     ret int 0
  }
}
```

- Familiar opcodes
- LLVM 1.0 C-style type system
- General syntax direction understood
- Bad ideas:
  - Constant pools
  - Classes
  - Encoding centric design

Need some code to build

- Picked GCC 3.0 as the first front-end:
  - USENIX: “GCC 3.0: The State of the Source” by Mark Mitchell
  - Didn’t support Java!

https://www.usenix.org/conference/als-2000/gcc-30-state-source

Need some code to build

- Picked GCC 3.0 as the first front-end:
  - USENIX: “GCC 3.0: The State of the Source” by Mark Mitchell
  - Didn’t support Java!
- Started work on RTL backend that produced LLVM IR
  - The first llvm-gcc!
  - llvm-gcc 3.4, 4.0, 4.2 and dragonegg came later

https://www.usenix.org/conference/als-2000/gcc-30-state-source

Version Control!

June 2001, 6 months later

svn co -r2 'http://llvm.org/svn/llvm-project/llvm/trunk' llvm-v1

LLVM v1

- Looks more similar to today's LLVM:

```
%pointer = type int *

implementation

int "test function"(int %i0, int %j0)
begin
    %array0 = malloc [4 x ubyte]              ; yields {[4 x ubyte]*}:array0
    %size   = add uint 2, 2                   ; yields {uint}:size = uint %4
    %array1 = malloc [ubyte], uint 4          ; yields {[ubyte]*}:array1
    %array2 = malloc [ubyte], uint %size      ; yields {[ubyte]*}:array2
    free [4x ubyte]* %array0
    free [ubyte]* %array1
    free [ubyte]* %array2


    alloca [ubyte], uint 5
    %ptr = alloca int                         ; yields {int*}:ptr
    store int* %ptr, int 3                    ; yields {void}
    %val = load int* %ptr                     ; yields {int}:val = int %3

    ret int 3
end
```

LLVM v1

- Looks more similar to today's LLVM:

```
%pointer = type int *

implementation

int "test function"(int %i0, int %j0)
begin
    %array0 = malloc [4 x ubyte]            ; yields {[4 x ubyte]*}:array0
    %size   = add uint 2, 2                 ; yields {uint}:size = uint %4
    %array1 = malloc [ubyte], uint 4        ; yields {[ubyte]*}:array1
    %array2 = malloc [ubyte], uint %size    ; yields {[ubyte]*}:array2
    free [4x ubyte]* %array0
    free [ubyte]* %array1
    free [ubyte]* %array2


    alloca [ubyte], uint 5
    %ptr = alloca int                       ; yields {int*}:ptr
    store int* %ptr, int 3                  ; yields {void}
    %val = load int* %ptr                   ; yields {int}:val = int %3

    ret int 3
end
```

LLVM v1

- Looks more similar to today's LLVM:

```
%pointer = type int *

implementation

int "test function"(int %i0, int %j0)
begin
    %array0 = malloc [4 x ubyte]              ; yields {[4 x ubyte]*}:array0
    %size   = add uint 2, 2                   ; yields {uint}:size = uint %4
    %array1 = malloc [ubyte], uint 4          ; yields {[ubyte]*}:array1
    %array2 = malloc [ubyte], uint %size      ; yields {[ubyte]*}:array2
    free [4x ubyte]* %array0
    free [ubyte]* %array1
    free [ubyte]* %array2


    alloca [ubyte], uint 5
    %ptr = alloca int                         ; yields {int*}:ptr
    store int* %ptr, int 3                    ; yields {void}
    %val = load int* %ptr                     ; yields {int}:val = int %3

    ret int 3
end
```

LLVM v1

- Looks more similar to today's LLVM:

```
%pointer = type int *

implementation

int "test function"(int %i0, int %j0)
begin
    %array0 = malloc [4 x ubyte]              ; yields {[4 x ubyte]*}:array0
    %size   = add uint 2, 2                   ; yields {uint}:size = uint %4
    %array1 = malloc [ubyte], uint 4          ; yields {[ubyte]*}:array1
    %array2 = malloc [ubyte], uint %size      ; yields {[ubyte]*}:array2
    free [4x ubyte]* %array0
    free [ubyte]* %array1
    free [ubyte]* %array2


    alloca [ubyte], uint 5
    %ptr = alloca int                         ; yields {int*}:ptr
    store int* %ptr, int 3                    ; yields {void}
    %val = load int* %ptr                     ; yields {int}:val = int %3

    ret int 3
end
```

LLVM v1

- Looks more similar to today's LLVM:

```
%pointer = type int *

implementation

int "test function"(int %i0, int %j0)
begin
    %array0 = malloc [4 x ubyte]              ; yields {[4 x ubyte]*}:array0
    %size   = add uint 2, 2                   ; yields {uint}:size = uint %4
    %array1 = malloc [ubyte], uint 4          ; yields {[ubyte]*}:array1
    %array2 = malloc [ubyte], uint %size      ; yields {[ubyte]*}:array2
    free [4x ubyte]* %array0
    free [ubyte]* %array1
    free [ubyte]* %array2


    alloca [ubyte], uint 5
    %ptr = alloca int                         ; yields {int*}:ptr
    store int* %ptr, int 3                    ; yields {void}
    %val = load int* %ptr                     ; yields {int}:val = int %3

    ret int 3
end
```

Lots of progress in 6 months

Lots of progress in 6 months

- LangRef.html

Lots of progress in 6 months

- LangRef.html
- Naming bikesheds painted: Class → Module, as → llvm-as, etc

Lots of progress in 6 months

- LangRef.html
- Naming bikesheds painted: Class → Module, as → llvm-as, etc
- Supported arrays, pointers, structs, some simd vectors

Lots of progress in 6 months

- LangRef.html
- Naming bikesheds painted: Class → Module, as → llvm-as, etc
- Supported arrays, pointers, structs, some simd vectors
- Call was documented (with invoke-like exception model!)

Lots of progress in 6 months

- LangRef.html
- Naming bikesheds painted: Class → Module, as → llvm-as, etc
- Supported arrays, pointers, structs, some simd vectors
- Call was documented (with invoke-like exception model!)
- New optimizations:

Lots of progress in 6 months

- LangRef.html
- Naming bikesheds painted: Class → Module, as → llvm-as, etc
- Supported arrays, pointers, structs, some simd vectors
- Call was documented (with invoke-like exception model!)
- New optimizations:
  - lib/Transforms/Scalar, lib/Transforms/IPO

Lots of progress in 6 months

- LangRef.html
- Naming bikesheds painted: Class → Module, as → llvm-as, etc
- Supported arrays, pointers, structs, some simd vectors
- Call was documented (with invoke-like exception model!)
- New optimizations:
  - lib/Transforms/Scalar, lib/Transforms/IPO
- IR verifier implemented

2001 - Getting the basics in place

- June 6 - First revision in CVS
- July 8 - getelementptr!
- July 15 - Vikram starts working on “llc” for SPARC
- Nov 16, 2001 - First paper submitted to PLDI

2002 - Faster progress

2002 - Faster progress

- January - Pass, PassManager, Analysis passes
- March - Data Structure Analysis (DSA)
- Summer - Mid-level optimizations
- September - Vikram teaches first class based on LLVM
  - llvm-commits and llvmdev come alive
- October - LLVM JIT and X86 target
- November - Bugpoint

2002 - Faster progress

- January - Pass, PassManager, Analysis passes
- March - Data Structure Analysis (DSA)
- Summer - Mid-level optimizations
- September - Vikram teaches first class based on LLVM
  - llvm-commits and llvmdev come alive
- October - LLVM JIT and X86 target
- November - Bugpoint
- December - Chris finishes master’s thesis on LLVM
  - “LLVM: An Infrastructure for Multi-Stage Optimization”

LLVM 1.0

October 24, 2003

http://llvm.org/releases/download.html#1.0

What did it do?

- Sparc, X86, and C Backend
- llvm-gcc: “3.4-llvm 20030827 (experimental)”
- Worked: SPEC CPU2000, Olden, Ptrdist, …
- 125K lines of code

What's New?

This is the first public release of the LLVM compiler infrastructure. As such, it is all new! In particular, we are providing a stable C compiler, beta C++ compiler, a C back-end, stable X86 and Sparc V9 static and JIT code generators, as well as a large suite of scalar and interprocedural optimizations.

The default optimizer sequence used by the C/C++ front-ends is:

1. CFG simplification (-simplifycfg)
2. Interprocedural dead code elimination (-globaldce)
3. Interprocedural constant propagation (-ipconstprop)
4. Dead argument elimination (-deadargelim)
5. Exception handling pruning (-prune-eh)
6. Function inlining (-inline)
7. Instruction combining (-instcombine)
8. Cast elimination (-raise)
9. Tail duplication (-tailduplicate)
10. CFG simplification (-simplifycfg)
11. Scalar replacement of aggregates (-scalarrepl)
12. Tail call elimination (-tailcallelim)
13. Instruction combining (-instcombine)
14. Reassociation (-reassociate)
15. Instruction combining (-instcombine)
16. CFG simplification (-simplifycfg)
17. Loop canonicalization (-loopsimplify)
18. Loop invariant code motion, with scalar promotion (-licm)
19. Global common subexpression elimination, with load elimination (-gcse)
20. Sparse conditional constant propagation (-sccp)
21. Instruction combining (-instcombine)
22. Induction variable canonicalization (-indvars)
23. Aggressive dead code elimination (-adce)
24. CFG simplication (-simplifycfg)
25. Dead type elimination (-deadtypeelim)
26. Global constant merging (-constmerge)

At link-time, the following optimizations are run:

1. Global constant merging (-constmerge)
2. [optional] Internalization [which marks most functions and global variables static] (-internalize)
3. Interprocedural constant propagation (-ipconstprop)
4. Interprocedural dead argument elimination (-deadargelim)
5. Instruction combining (-instcombine)
6. CFG simplification (-simplifycfg)
7. Interprocedural dead code elimination (-globaldce)

What did it do?

- Sparc, X86, and C Backend
- llvm-gcc: “3.4-llvm 20030827 (experimental)”
- Worked: SPEC CPU2000, Olden, Ptrdist, …
- 125K lines of code
- UIUC/BSD License
  - Wanted the code to be **used**
  - Even commercially
  - No barriers for adoption

What's New?

This is the first public release of the LLVM compiler infrastructure. As such, it is all new! In particular, we are providing a stable C compiler, beta C++ compiler, a C back-end, stable X86 and Sparc V9 static and JIT code generators, as well as a large suite of scalar and interprocedural optimizations.

The default optimizer sequence used by the C/C++ front-ends is:

1. CFG simplification (-simplifycfg)
2. Interprocedural dead code elimination (-globaldce)
3. Interprocedural constant propagation (-ipconstprop)
4. Dead argument elimination (-deadargelim)
5. Exception handling pruning (-prune-eh)
6. Function inlining (-inline)
7. Instruction combining (-instcombine)
8. Cast elimination (-raise)
9. Tail duplication (-tailduplicate)
10. CFG simplification (-simplifycfg)
11. Scalar replacement of aggregates (-scalarrepl)
12. Tail call elimination (-tailcallelim)
13. Instruction combining (-instcombine)
14. Reassociation (-reassociate)
15. Instruction combining (-instcombine)
16. CFG simplification (-simplifycfg)
17. Loop canonicalization (-loopsimplify)
18. Loop invariant code motion, with scalar promotion (-licm)
19. Global common subexpression elimination, with load elimination (-gcse)
20. Sparse conditional constant propagation (-sccp)
21. Instruction combining (-instcombine)
22. Induction variable canonicalization (-indvars)
23. Aggressive dead code elimination (-adce)
24. CFG simplication (-simplifycfg)
25. Dead type elimination (-deadtypeelim)
26. Global constant merging (-constmerge)

At link-time, the following optimizations are run:

1. Global constant merging (-constmerge)
2. [optional] Internalization [which marks most functions and global variables static] (-internalize)
3. Interprocedural constant propagation (-ipconstprop)
4. Interprocedural dead argument elimination (-deadargelim)
5. Instruction combining (-instcombine)
6. CFG simplification (-simplifycfg)
7. Interprocedural dead code elimination (-globaldce)

1.0 Limitations

- Completely unsupported:
  - vectors, inline asm, complex numbers, exception handling, …
  - debug info
  - structs with more than 256 fields

1.0 Limitations

- Completely unsupported:
  - vectors, inline asm, complex numbers, exception handling, …
  - debug info
  - structs with more than 256 fields
- Tons of bugs

1.0 Limitations

- Completely unsupported:
  - vectors, inline asm, complex numbers, exception handling, …
  - debug info
  - structs with more than 256 fields
- Tons of bugs
- Instcombine was only 2000 LOC!

LLVM 1.0 IR

```
%node_t = type { double*, %node_t*, %node_t**, double**, double*, int, int }

void %localize_local(%node_t* %nodelist) {
bb0:
        %nodelist = alloca %node_t*
        store %node_t* %nodelist, %node_t** %nodelist
        br label %bb1

bb1:
        %reg107 = load %node_t** %nodelist
        %cond211 = seteq %node_t* %reg107, null
        br bool %cond211, label %bb3, label %bb2

bb2:
        %reg109 = phi %node_t* [ %reg110, %bb2 ], [ %reg107, %bb1 ]
        %reg212 = getelementptr %node_t* %reg109, long 0, ubyte 1
        %reg110 = load %node_t** %reg212
        %cond213 = setne %node_t* %reg110, null
        br bool %cond213, label %bb2, label %bb3

bb3:
        ret void
}
```

LLVM 1.0 IR

```
%node_t = type { double*, %node_t*, %node_t**, double**, double*, int, int }

void %localize_local(%node_t* %nodelist) {
bb0:
        %nodelist = alloca %node_t*
        store %node_t* %nodelist, %node_t** %nodelist
        br label %bb1

bb1:
        %reg107 = load %node_t** %nodelist
        %cond211 = seteq %node_t* %reg107, null
        br bool %cond211, label %bb3, label %bb2

bb2:
        %reg109 = phi %node_t* [ %reg110, %bb2 ], [ %reg107, %bb1 ]
        %reg212 = getelementptr %node_t* %reg109, long 0, ubyte 1
        %reg110 = load %node_t** %reg212
        %cond213 = setne %node_t* %reg110, null
        br bool %cond213, label %bb2, label %bb3

bb3:
        ret void
}
```

LLVM 1.0 IR

```
%node_t = type { double*, %node_t*, %node_t**, double**, double*, int, int }

void %localize_local(%node_t* %nodelist) {
bb0:
        %nodelist = alloca %node_t*
        store %node_t* %nodelist, %node_t** %nodelist
        br label %bb1

bb1:
        %reg107 = load %node_t** %nodelist
        %cond211 = seteq %node_t* %reg107, null
        br bool %cond211, label %bb3, label %bb2

bb2:
        %reg109 = phi %node_t* [ %reg110, %bb2 ], [ %reg107, %bb1 ]
        %reg212 = getelementptr %node_t* %reg109, long 0, ubyte 1
        %reg110 = load %node_t** %reg212
        %cond213 = setne %node_t* %reg110, null
        br bool %cond213, label %bb2, label %bb3

bb3:
        ret void
}
```

LLVM 1.0 IR

```
%node_t = type { double*, %node_t*, %node_t**, double**, double*, int, int }

void %localize_local(%node_t* %nodelist) {
bb0:
        %nodelist = alloca %node_t*
        store %node_t* %nodelist, %node_t** %nodelist
        br label %bb1

bb1:
        %reg107 = load %node_t** %nodelist
        %cond211 = seteq %node_t* %reg107, null
        br bool %cond211, label %bb3, label %bb2

bb2:
        %reg109 = phi %node_t* [ %reg110, %bb2 ], [ %reg107, %bb1 ]
        %reg212 = getelementptr %node_t* %reg109, long 0, ubyte 1
        %reg110 = load %node_t** %reg212
        %cond213 = setne %node_t* %reg110, null
        br bool %cond213, label %bb2, label %bb3

bb3:
        ret void
}
```

LLVM 1.0 IR

```
%node_t = type { double*, %node_t*, %node_t**, double**, double*, int, int }

void %localize_local(%node_t* %nodelist) {
bb0:
        %nodelist = alloca %node_t*
        store %node_t* %nodelist, %node_t** %nodelist
        br label %bb1

bb1:
        %reg107 = load %node_t** %nodelist
        %cond211 = seteq %node_t* %reg107, null
        br bool %cond211, label %bb3, label %bb2

bb2:
        %reg109 = phi %node_t* [ %reg110, %bb2 ], [ %reg107, %bb1 ]
        %reg212 = getelementptr %node_t* %reg109, long 0, ubyte 1
        %reg110 = load %node_t** %reg212
        %cond213 = setne %node_t* %reg110, null
        br bool %cond213, label %bb2, label %bb3

bb3:
        ret void
}
```

LLVM 1.0 IR

```
%node_t = type { double*, %node_t*, %node_t**, double**, double*, int, int }

void %localize_local(%node_t* %nodelist) {
bb0:
        %nodelist = alloca %node_t*
        store %node_t* %nodelist, %node_t** %nodelist
        br label %bb1

bb1:
        %reg107 = load %node_t** %nodelist
        %cond211 = seteq %node_t* %reg107, null
        br bool %cond211, label %bb3, label %bb2

bb2:
        %reg109 = phi %node_t* [ %reg110, %bb2 ], [ %reg107, %bb1 ]
        %reg212 = getelementptr %node_t* %reg109, long 0, ubyte 1
        %reg110 = load %node_t** %reg212
        %cond213 = setne %node_t* %reg110, null
        br bool %cond213, label %bb2, label %bb3

bb3:
        ret void
}
```

Tablegen .td Descriptions

Tablegen .td Descriptions

Sparc:

```
// Section A.18: Floating-Point Multiply and Divide - p165
def FMULS  : F3_16<2, 0b110100, 0b001001001, "fmuls">;
def FMULD  : F3_16<2, 0b110100, 0b001001010, "fmuld">;
def FMULQ  : F3_16<2, 0b110100, 0b001001011, "fmulq">;
def FSMULD : F3_16<2, 0b110100, 0b001101001, "fsmuld">;
def FDMULQ : F3_16<2, 0b110100, 0b001101110, "fdmulq">;
def FDIVS  : F3_16<2, 0b110100, 0b001001101, "fdivs">;
def FDIVD  : F3_16<2, 0b110100, 0b001001110, "fdivs">;
def FDIVQ  : F3_16<2, 0b110100, 0b001001111, "fdivs">;
```

Tablegen .td Descriptions

Sparc:

```
// Section A.18: Floating-Point Multiply and Divide - p165
def FMULS  : F3_16<2, 0b110100, 0b001001001, "fmuls">;
def FMULD  : F3_16<2, 0b110100, 0b001001010, "fmuld">;
def FMULQ  : F3_16<2, 0b110100, 0b001001011, "fmulq">;
def FSMULD : F3_16<2, 0b110100, 0b001101001, "fsmuld">;
def FDMULQ : F3_16<2, 0b110100, 0b001101110, "fdmulq">;
def FDIVS  : F3_16<2, 0b110100, 0b001001101, "fdivs">;
def FDIVD  : F3_16<2, 0b110100, 0b001001110, "fdivs">;
def FDIVQ  : F3_16<2, 0b110100, 0b001001111, "fdivs">;
```

X86:

```
// Arithmetic...
def ADDrr8   : I2A8 <"add", 0x00, MRMDestReg>,         Pattern<(set R8 , (plus R8 , R8 ))>;
def ADDrr16  : I2A16<"add", 0x01, MRMDestReg>, OpSize, Pattern<(set R16, (plus R16, R16))>;
def ADDrr32  : I2A32<"add", 0x01, MRMDestReg>,         Pattern<(set R32, (plus R32, R32))>;
def ADDri8   : I2A8 <"add", 0x80, MRMS0r    >,         Pattern<(set R8 , (plus R8 , imm))>;
def ADDri16  : I2A16<"add", 0x81, MRMS0r    >, OpSize, Pattern<(set R16, (plus R16, imm))>;
def ADDri32  : I2A32<"add", 0x81, MRMS0r    >,         Pattern<(set R32, (plus R32, imm))>;
def ADDri16b : I2A8 <"add", 0x83, MRMS0r    >, OpSize;
def ADDri32b : I2A8 <"add", 0x83, MRMS0r    >;
```

CREDITS.TXT

11 People, including:

```
N: Vikram Adve
D: The Sparc64 backend, provider of much wisdom, and motivator for LLVM

N: Tanya Lattner
D: The llvm-ar tool

N: John T. Criswell
D: Autoconf support, QMTest database, documentation improvements

N: Chris Lattner
D: Primary architect of LLVM

N: Bill Wendling
D: The `Lower Setjmp/Longjmp' pass, improvements to the -lowerswitch pass.
```

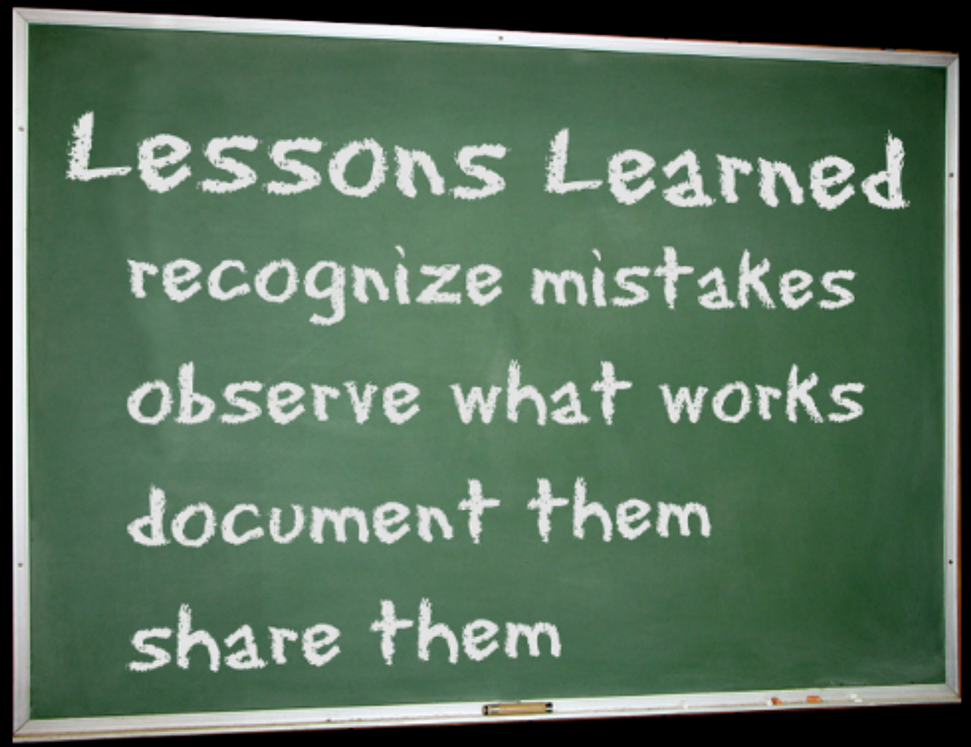
LLVM 3.4 coming soon!

10 years and 23 releases later

http://llvm.org/releases/

Lessons learned

- Gap between interesting ideas and “production quality”
- Continuous improvement, not perfection
- Persistence and dedication required
- Go deep, not broad
- Have smarter people rewrite your code



A great community makes it possible!

A great community makes it possible!

12 Month Summary

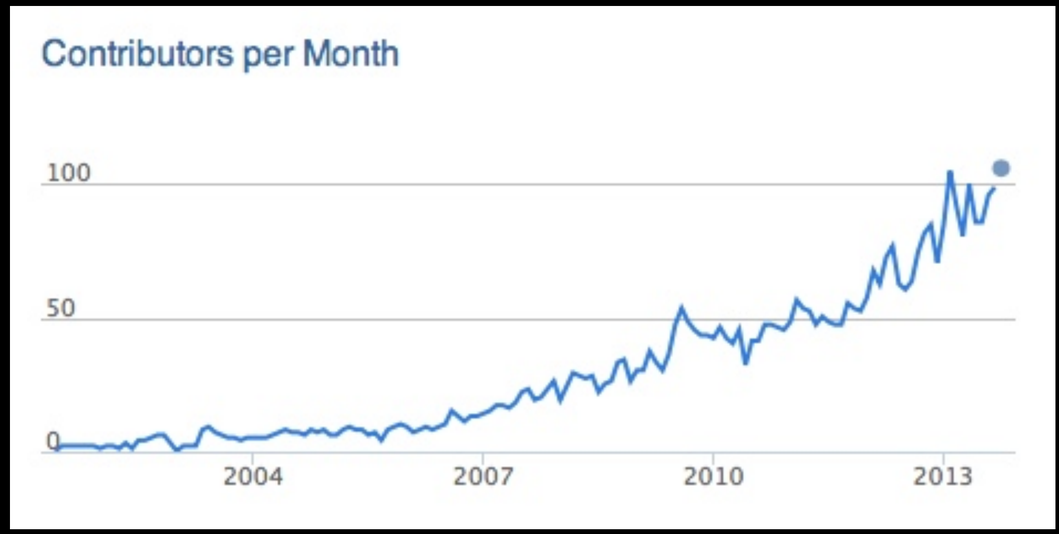
Oct 31 2012 — Oct 31 2013

12706 Commits

Up +2917 (29%) from previous 12 months

226 Contributors

Up +59 (35%) from previous 12 months



A great community makes it possible!

12 Month Summary

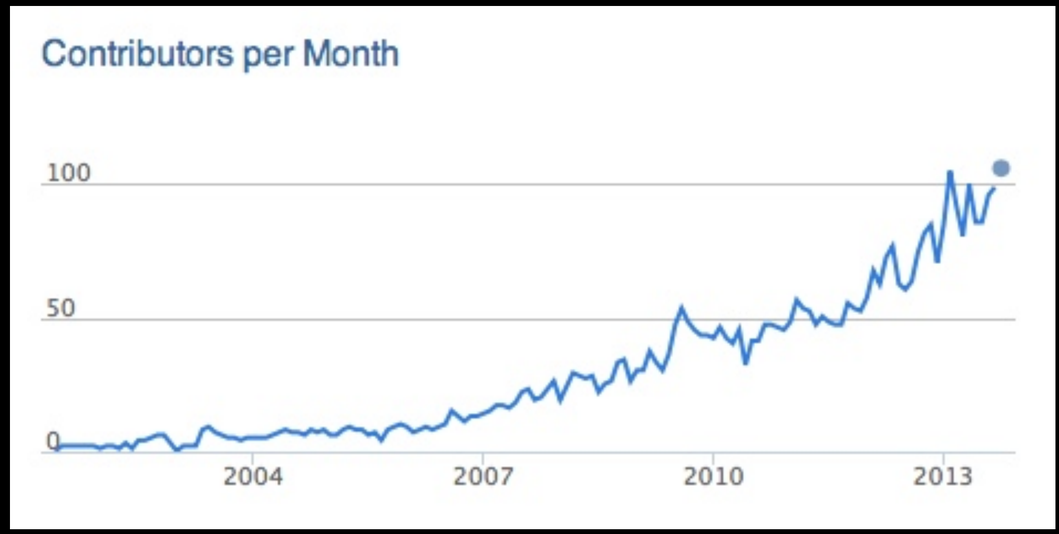
Oct 31 2012 — Oct 31 2013

12706 Commits

Up +2917 (29%) from previous 12 months

226 Contributors

Up +59 (35%) from previous 12 months



Thank you all!