

SABLEJIT: A Retargetable Just-In-Time Compiler for a Portable Virtual Machine

David Bélanger

dbelan2@cs.mcgill.ca

Sable Research Group

McGill University

Montreal, QC

April 25, 2003

Overview

- Introduction and Motivation
- SableVM
- Design Overview
- At runtime
 - Compile and Call
 - Exception Handling
 - Garbage Collection
- Frontend / Backend
- Self-compilation
- Partial Compilation
- Conclusion and Further Work

Introduction and Motivation

- SableVM is designed to be an efficient portable bytecode interpreter.
- Code interpretation has a significant overhead and just-in-time (JIT) compilers are designed to overcome this problem.
- Most JITs have focused on efficiency, making code very machine specific. Retargeting to a new platform involves a lot of work.
- A compilation unit is usually a method.
- SABLEJIT is designed to be relatively easy to port to new platforms while remaining reasonably efficient.
- Furthermore, a serious attempt is made to allow flexibility concerning the compilation unit.

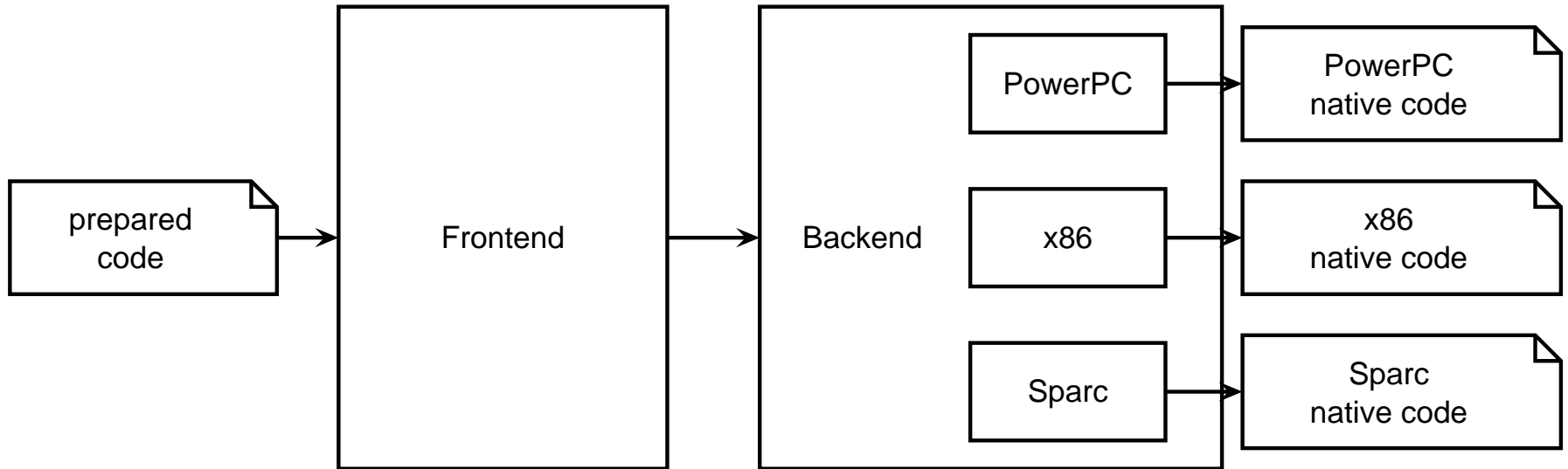
SableVM

- Portable bytecode interpreter written by Etienne Gagnon.
- 3 interpreters “flavours”:
 - Switch interpreter.
 - Direct-threaded interpreter.
 - Inlined-threaded interpreter.
- SABLEJIT compiles the prepared code used by the switch interpreter.
 - Array of words, not bytes.
 - Some bytecodes split: GETFIELD_INT, GETFIELD_BOOLEAN, ...
 - Preparation sequences at the end.

Design Overview

- SABLEJIT is written in Java.
- SABLEJIT can self-compile.
- One-pass code generator (1 pass + patching branches)
 - No IR data structure built.
 - Code generated right away.
 - Architecture dependent aspects simply abstracted.
 - Considering building a LIR.
- Both Linux/PPC and Linux/x86 supported.
- Should be relatively easy to add support for new architectures (especially RISC)

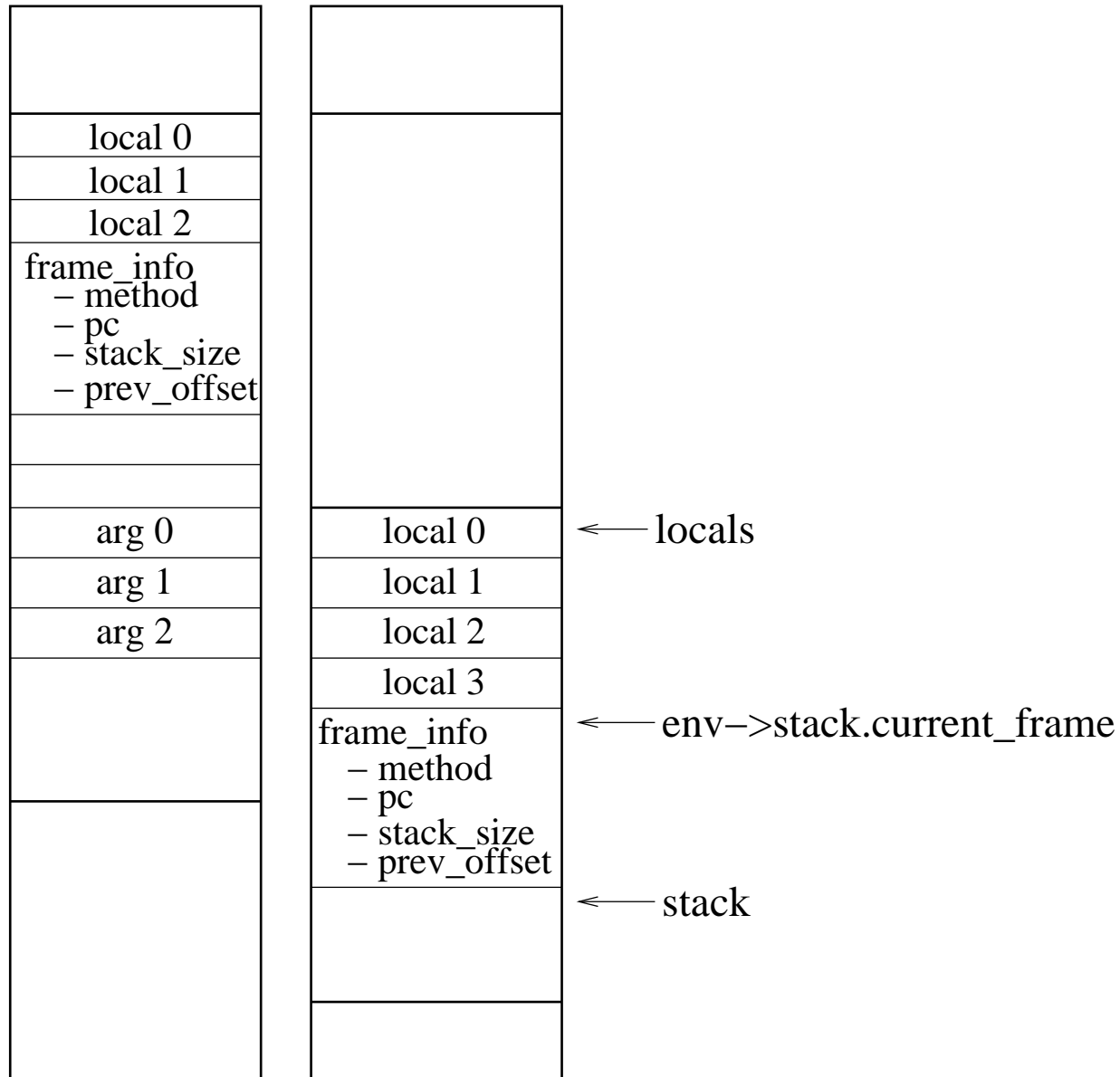
Design Overview



Design Overview

- Both SableVM and SABLEJIT shares the same Java stack.
 - Locals are read/written from/to the stack.
 - Return value is written directly to the stack.
- Simplifies handling of exceptions and garbage collection significantly.
- SABLEJIT is in fact mimicking SableVM.

SableVM Java Stack



Compilation

```
switch (bytecode) {  
  
    case INVOKEVIRTUAL:  
        if (should compile) {  
            COMPILE  
        }  
  
        /* setup stack frame */  
  
        if (method is compiled) {  
            CALLCOMPILEDPCODE  
        }  
  
}
```

Compilation

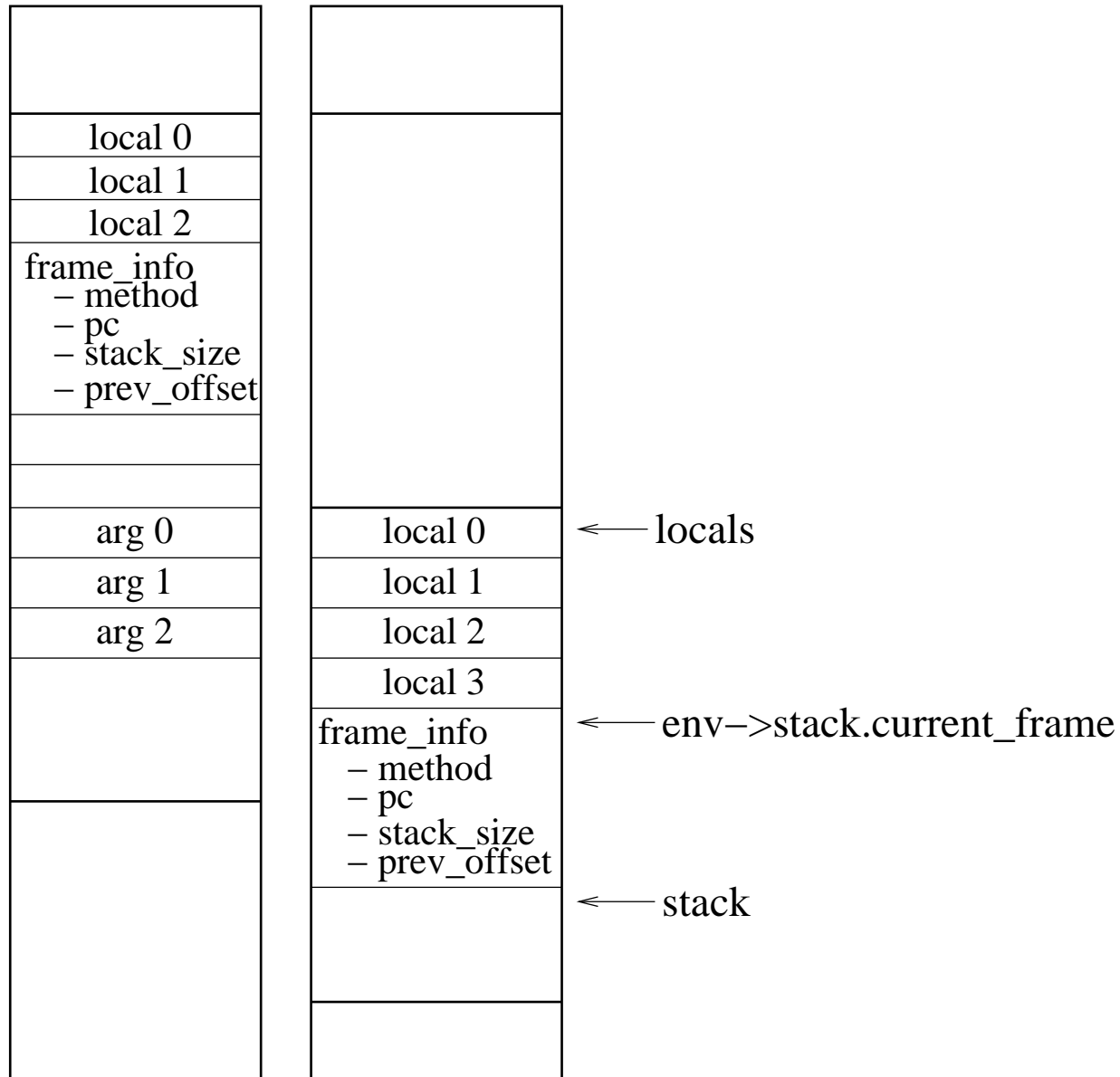
- SableVM invokes `Compiler.compile(int[] bytecode, int[] linkVector)` where
 - `bytecode` is the array of prepared code (used by the switch interpreter)
 - `linkVector` array of pointers to data structures and functions in SableVM.
- This method returns the compiled code array.
- Note: SABLEJIT is robust, if an exception occurs during compilation:
 - Method is marked as uncompileable
 - Method is interpreted instead

Calling Compiled Code

```
switch (bytecode) {  
  
    case INVOKEVIRTUAL:  
        /* setup stack frame */  
  
        if (method is compiled) {  
            stack_inc = compiled_code(env, locals, stack);  
            stack_size += stack_inc  
        }  
    }  
}
```

- `env` - `_svmt_JNIEnv *env` (thread context)
- `locals` - pointer to locals on Java stack
- `stack` - pointer to expression stack

Java Stack



Exception Handling - SABLEJIT

- Solution: Since we are using the same stack as the interpreter, let SableVM handle the exceptions.
- If an exception is thrown by the compiled code, the compiled code:
 - Set the exception object in `env->throwable`
 - Saves the bytecode pc in the stack frame.
 - Returns `-1`.
- Control will get back to interpreter.
- If the compiled code returns `-1`, the interpreter jumps to the exception handler.
- Note: The java stack is not popped by the compiled code.

Exception Handling - SableVM

```
switch (bytecode) {  
  
    case INVOKEVIRTUAL:  
  
        stack_inc = compile_code(...)  
        if (stack_inc == -1) {  
            goto exception_handler;  
        }  
  
}  
  
exception_handler:  
    /* Code to handle exception */
```

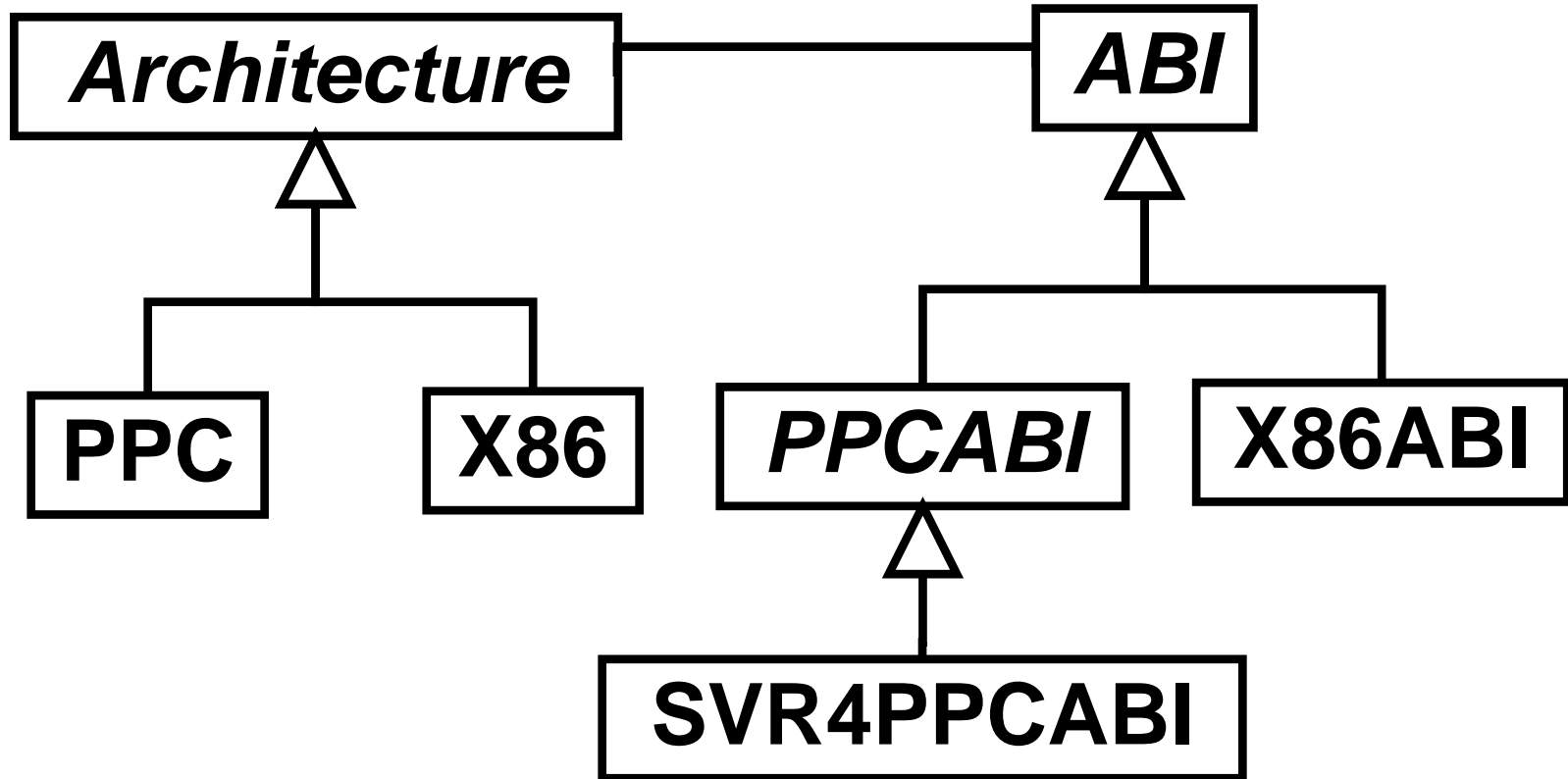
Garbage Collection

- Solution: Since we work directly on the Java stack, no need to maintain extra roots.
- Needs to store the state at some specific points where GC could be triggered.
 - Bytecode pc
 - Stack size
- These points same as interpreter.
- Compilation may trigger garbage collection.
- General idea is simple but difficult to debug.

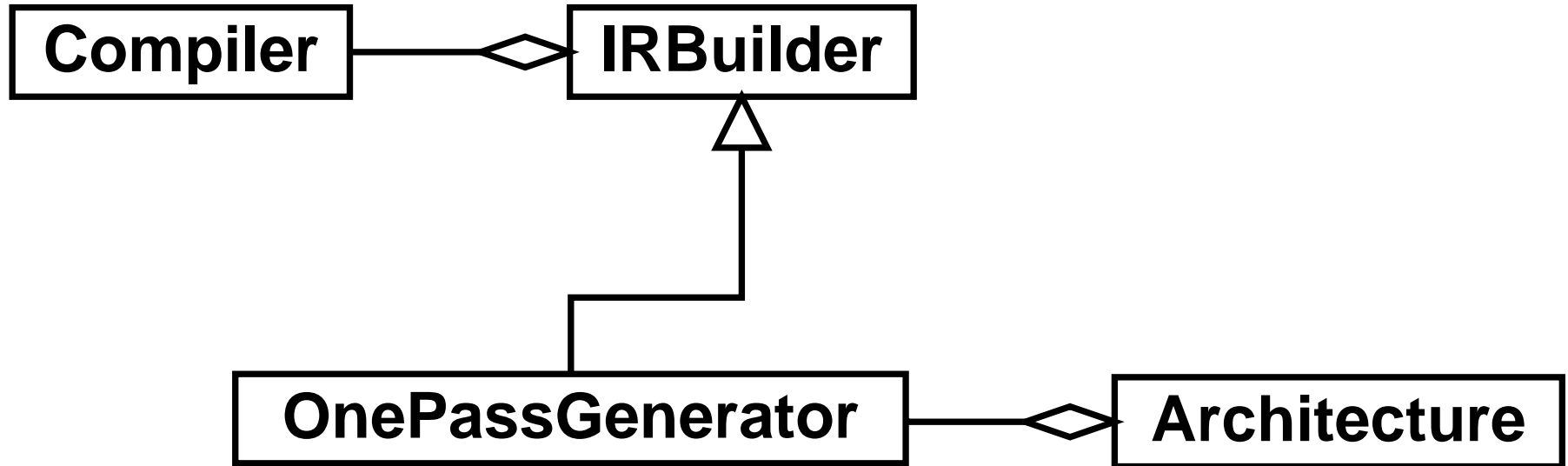
Backend - Instruction Set

- Concept similar to `VCODE`.
- RISC-like instruction set (ex: `add rd, rs, rt`).
- Each instruction correspond to a method (ex: `addI(rd, rs, rt)`).
- Some instructions implemented in terms of others (ex: `mulI`).
 - Allows a quick port.
 - Override for efficiency.
- System independent branch resolution and patching.
- x86
 - CISC architecture (ex: `add rd, rs`)
 - Only 8 “GPR” registers.
 - Restrictions on register usage

Backend - Design

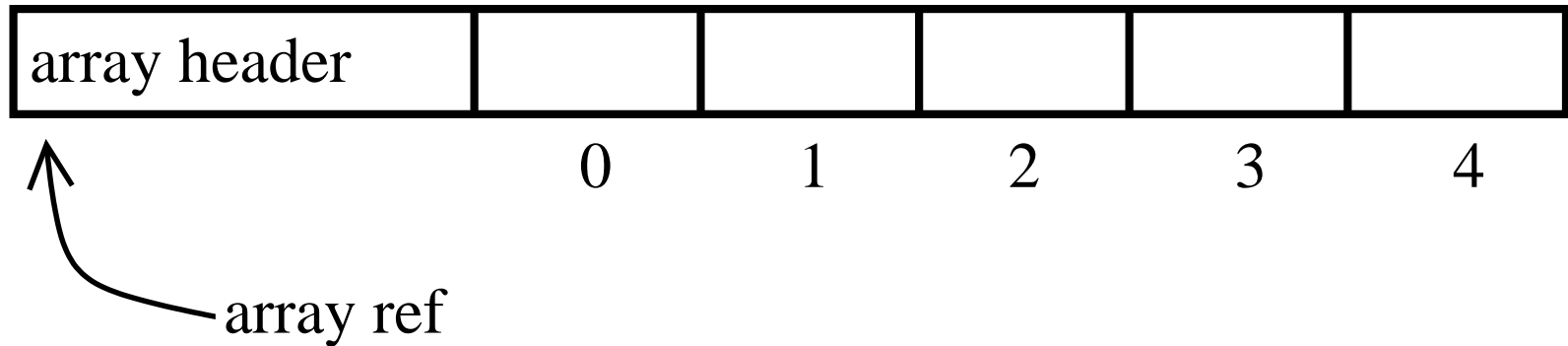


Frontend - Backend

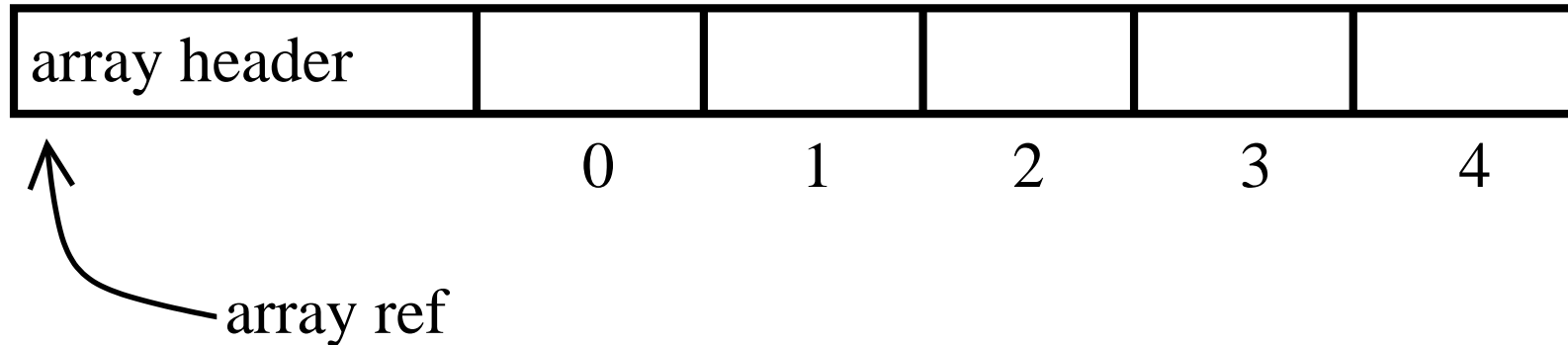


Example - iaload - Intro

- Integer Array Load (bytecode `iaload`)
stack effect: $arrayref\ index \rightarrow value$
result: $value = arrayref[index]$
- SableVM int array representation



Example - iaload - Steps



1. Load array reference
2. Load index
3. Check if reference not null
4. Check if index within array bounds
5. Compute address
6. Load element

Example - iaload - Code

```
// arrayref index --> value
public void build_iaload() throws SableJITException {
    pop(tmp2); // index
    pop(tmp1); // ref
    checkNullPointerException(tmp1);
    checkArrayIndexOutOfBoundsException(tmp1, tmp2);
    // advance pointer to first element
    arch.addiI(tmp1, tmp1, ALIGNED_ARRAY_INSTANCE_SIZE);
    arch.shlI(tmp2, tmp2, 2); // multiply index by 4
    arch.addI(tmp1, tmp1, tmp2); // find element
    arch.loadI(tmp1, 0, tmp1); // load element
    push(tmp1); // push it on stack
}
```

Porting Approach

- Instead of taking the approach:
“To port to a new platform, simply implement all the bytecodes”
We are taking the approach:
“To port to a new platform, simply implement `load`, `store`, `add`, ...”
- For example, porting to a new platform does not require knowing the layout of objects, arrays, fields etc.
- A peephole optimizer could be used to optimize these sequence of operations.

Backend - Calling Conventions

- SableVM needs to call compiled code.
- Sometimes, compiled code needs to call SableVM code.
- Different systems uses different calling conventions.
- Solution: Abstract calling convention
 - prologue() and epilogue()
 - moveArg(int rd, int n)
 - loadLocall(int rd, int index);
 - storeLocall(int rs, int index);
 - moveReturnValue(int rd);
 - pushRegArg(int type, int rs);
 - callReg(int rs);

Self-Compilation

- Allow the JIT to compile itself.

- Need to avoid compilation loops

```
invoke foo() - compilation triggered
```

```
  compile() is called
```

```
    begin()
```

```
      addiI()
```

```
        addiI() - compilation triggered
```

```
          compile()
```

```
            begin()
```

```
              addiI() already compiling, interpret
```

- Code needs to be re-entrant (several compilation in progress even if single-threaded)

- Using knowledge about itself: Inlining virtual invokes

Partial Compilation

- Example: Compiling hot loops

```
public int __compile_me_partial() {  
    int sum = 0;  
    for (int i = 0; i < 100000; i++) {  
        sum += i;  
    }  
    return sum;  
}
```

- Profile back edges
- Send code fragment to compiler
- Relatively easy to implement
- Trace-based JIT: Segments spanning several methods.

Conclusion and Further Work

- We have presented an overview of the current design of `SABLEJIT`.
- We have seen how `SABLEJIT` can be retargeted to new platforms.
- Further Work
 - Complete implementation (mostly `float`, `double` and `long` operations and synchronization).
 - Add a LIR for instructions and a peephole optimizer.
 - Complete implementation of virtual registers.
 - Add a customized memory manager.
 - Add profiling.
 - Measure performance.
 - Port to new platforms.