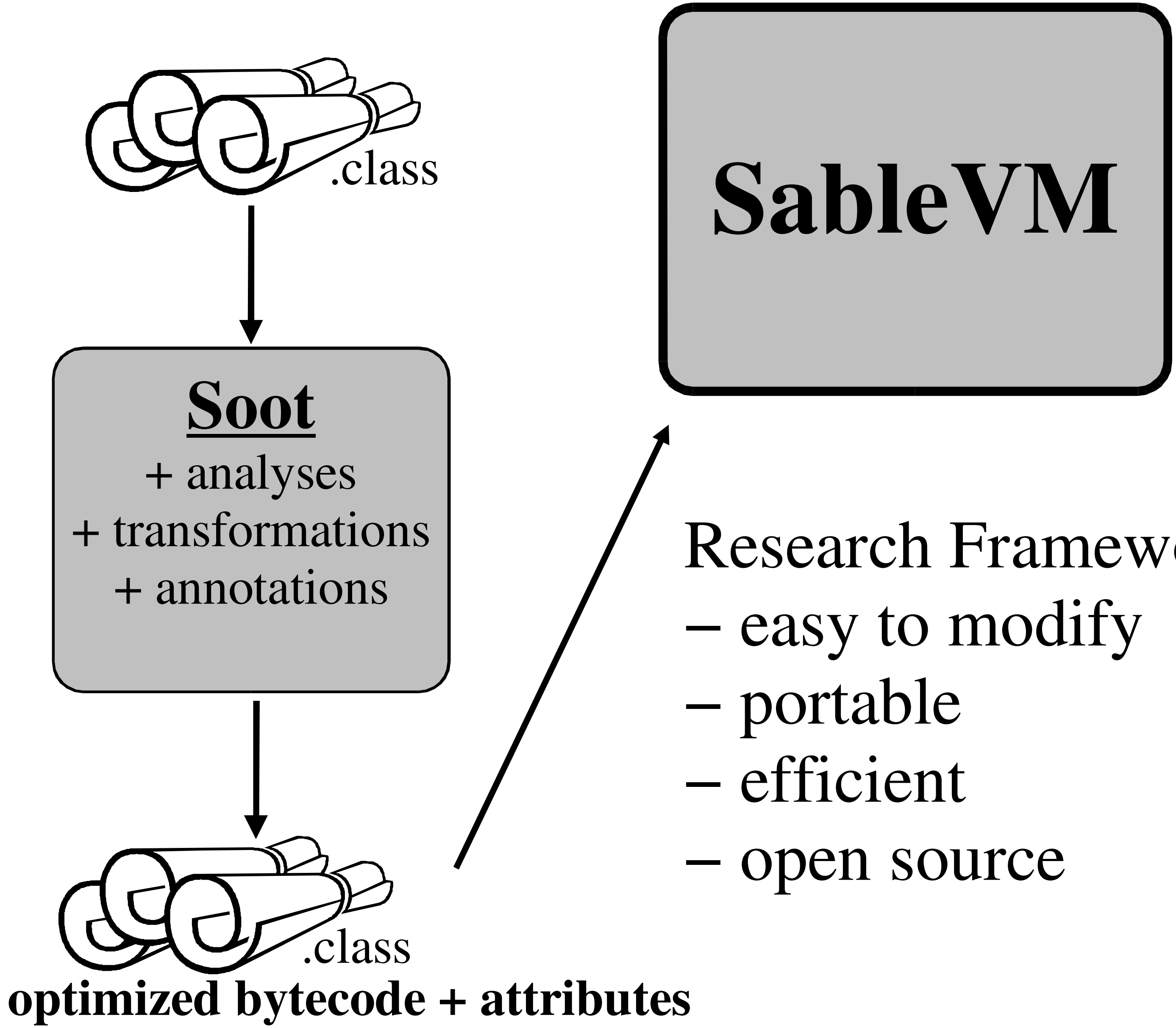
# SableVM: A Research Framework for the Efficient Execution of Java Bytecode

**Etienne Gagnon**
**Laurie Hendren**
Sable Research Group
McGill University

# Outline

- Motivation & Objectives
- Core Interpreter
- Bidirectional Object Instance Layout
- Sparse Interface Virtual Table Layout
- Busy Wait Free Thin Locks
- Experimental Results
- Conclusion

# Context: The Sable Project




Research Framework:
- easy to modify
- portable
- efficient
- open source

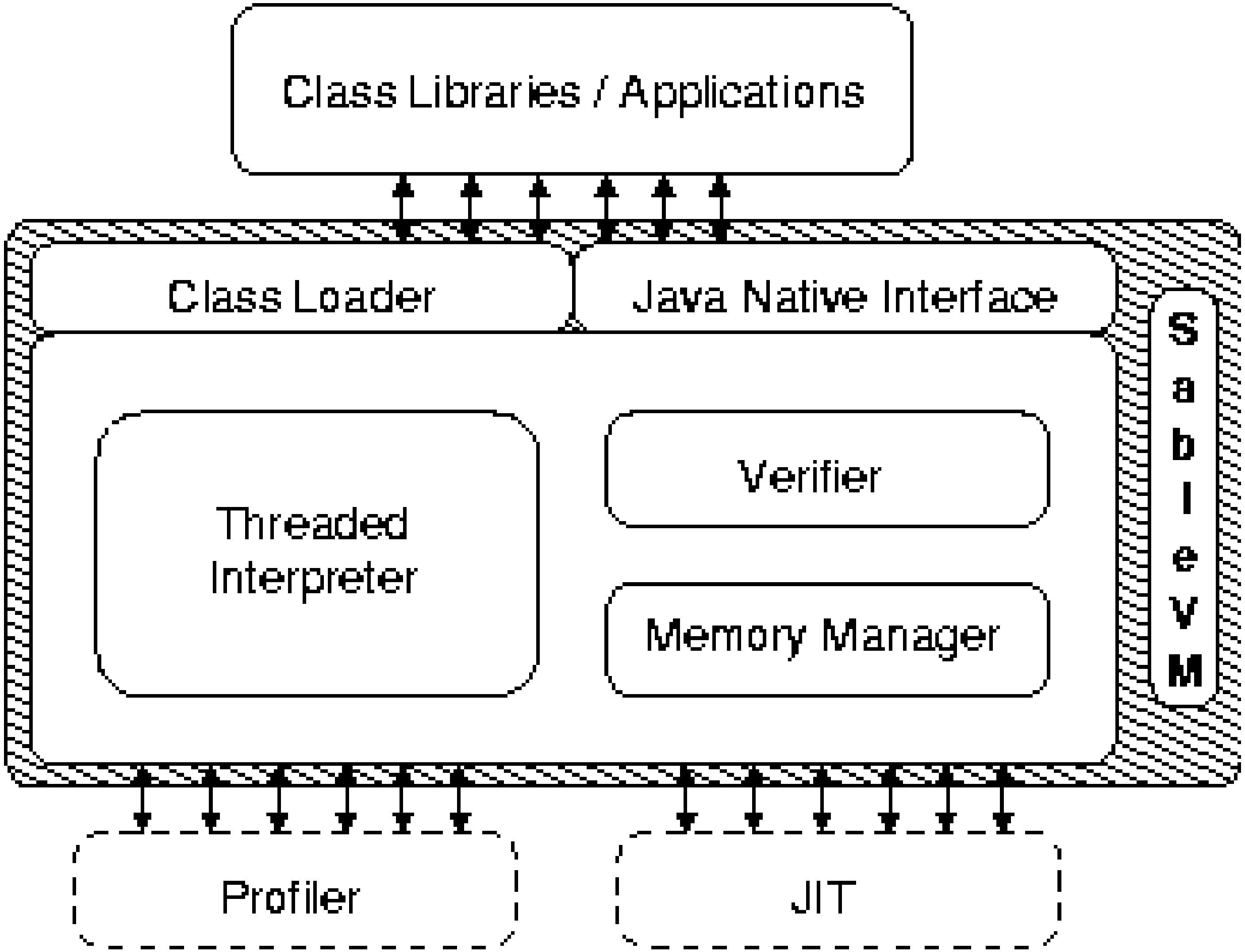
# Objectives

- Portable, yet efficient open–source JVM, useable as a research framework for the execution of Java bytecode
- Test performance enhancement techniques:
  - New object instance layout for faster garbage collection
  - New interface virtual table layout for interface method calls at the cost of normal virtual method calls
  - Eliminating busy wait on contention for *thin locks*
- **Note:** These techniques are meant to be useable within other JVMs, including those based on JIT compilation and dynamic feedback systems

# Advantages of an Efficient & Portable Interpreter

- It simplifies the implementation of new (experimental) features:
  - No assembly code to write
  - No dynamic compiler and/or feedback system to modify
- Can be used as the base interpreter in a hybrid system (interpreter/profiler/compiler)
- Relatively small at two levels (source code & binary code) because it does not include an embedded compiler

# SableVM Framework Overview



# Core Interpreter

- Written in portable ISO C with POSIX library
  - Except: *compare−and−swap*, optional *label as value*
- Interpreter in 3 flavors (through macro expansion):
  - Switch based
    (2 ind. Branches, ISO C)
  - Direct threaded
    (1 ind. Branch, *label as value*)
  - Selectively−inlined direct−threaded [Piumarta]
    (0 ind. branch within basic block, memory overhead)
- In all 3 flavors: Java bytecodes are first mapped to internal codes e.g:
  LDC –> LDC_INT, LDC_FLOAT, LDC_STRING

# Traditional *switch based* interpreter

```
int stack[10];
char code[] =
  {PUSH_1, PUSH_2, ADD, PRINT, END};

char *pc = &code[0];
int *sp = &stack[-1];

while (true) {
  switch (*pc++) {
    case PUSH_1: *++sp = 1; break;
    case PUSH_2: *++sp = 2; break;
    case ADD: {
      int tmp = *sp--;
      *sp = tmp + *sp;} break;
    case PRINT: printf("%d\n",*sp--);break;
    case END: goto end_label;
}
end_label:
```

# Direct–threaded interpreter

```
int stack[10];
void *code[] =  /* GNU C only! */
  {&&PUSH_1,&&PUSH_2,&&ADD,&&PRINT,&&END};

void **pc = &code[0];
int *sp = &stack[-1];

goto **(pc++);

PUSH_1: *++sp = 1; goto **(pc++);
PUSH_2: *++sp = 2; goto **(pc++);
ADD: {
  int tmp = *sp--;
  *sp = tmp + *sp;} goto **(pc++);
PRINT: printf("%d\n",*sp--); goto **(pc++);
END: goto end_label;
}
end_label:
```

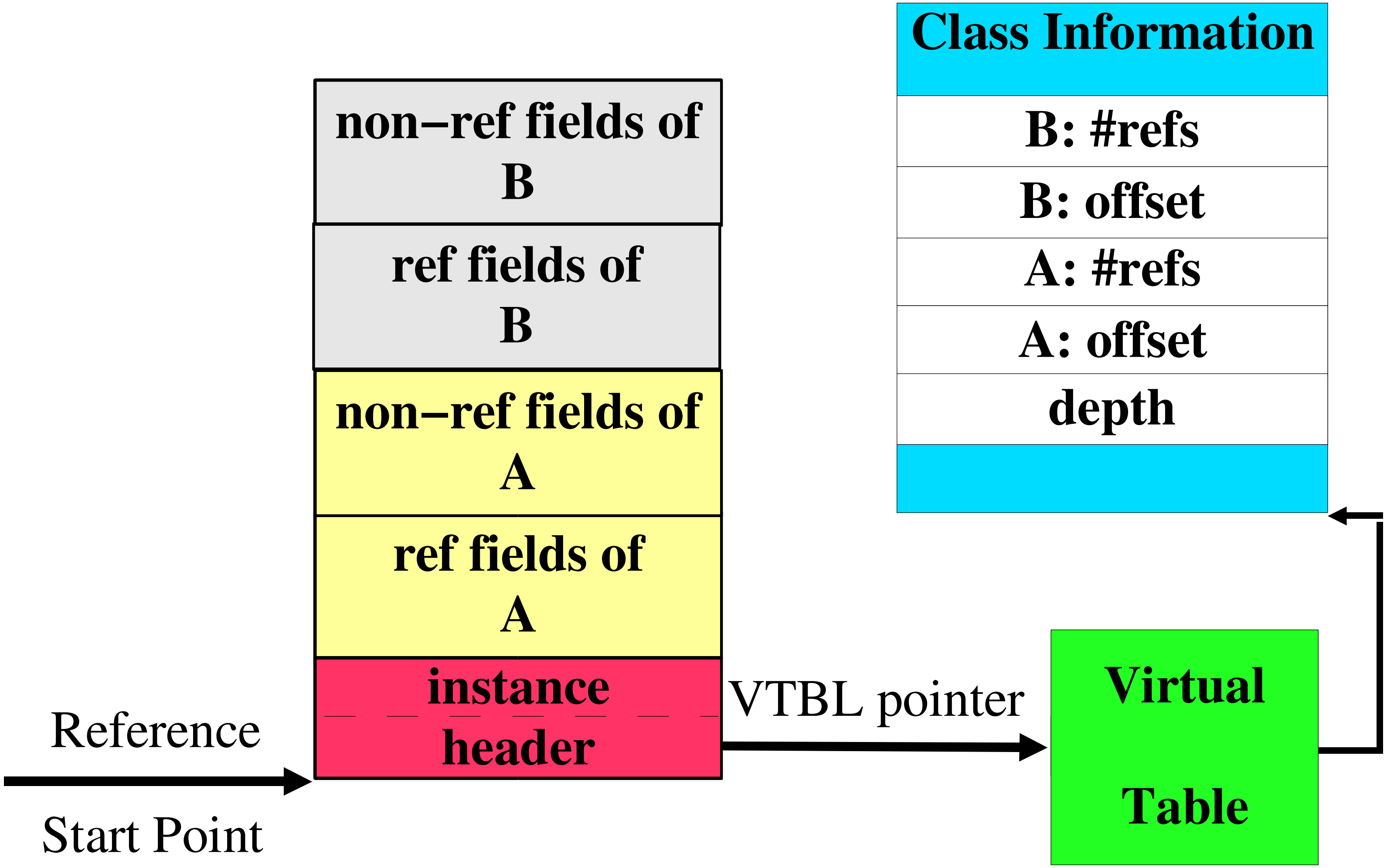
# New High–Level Performance Enhancement Techniques

- NOTE: VM implementation independent
- Contributed techniques:
  - Bidirectional object layout for faster garbage collection
  - Interface method lookup at the cost of virtual method lookup
  - Thin locks without busy–wait

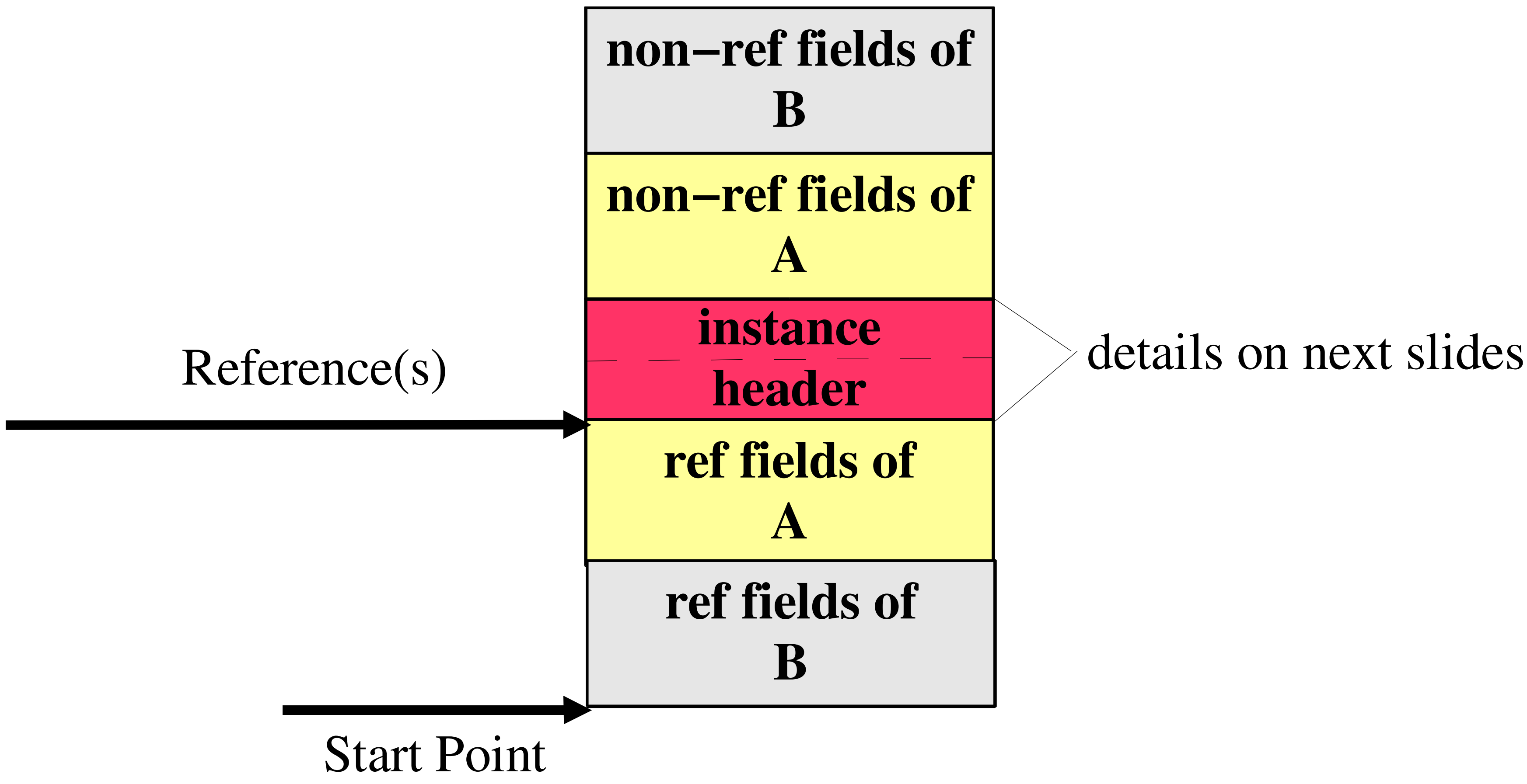
# Bidirectional Object Layout (for fast GC tracing)

- Main idea: Group all references together in instance layout
- How?
  - Grow objects in two directions
  - Put references before object header
- Properties
  - **References** point to the middle of objects
  - Object layout in memory **starts** with outer reference

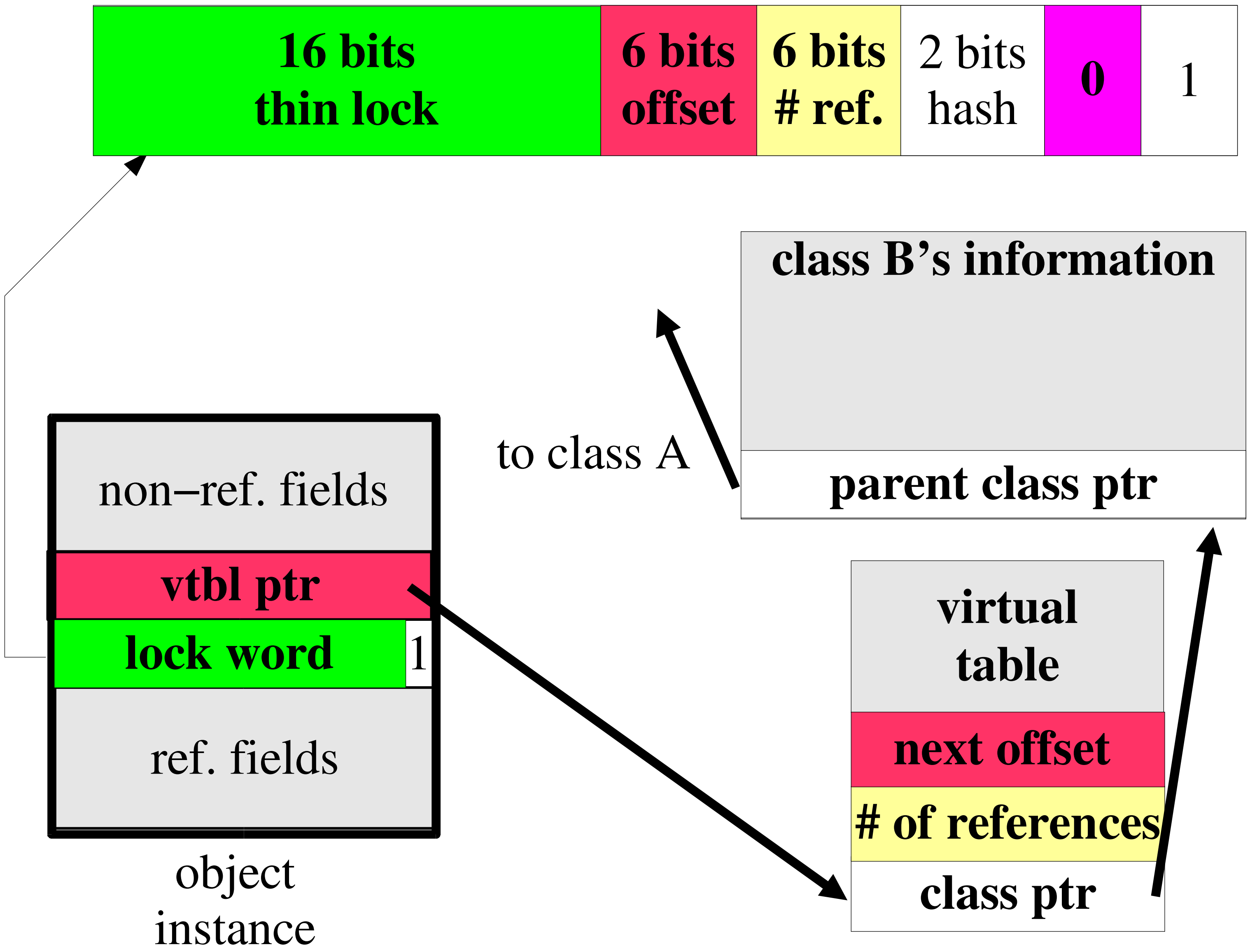
# Traditional Instance Layout



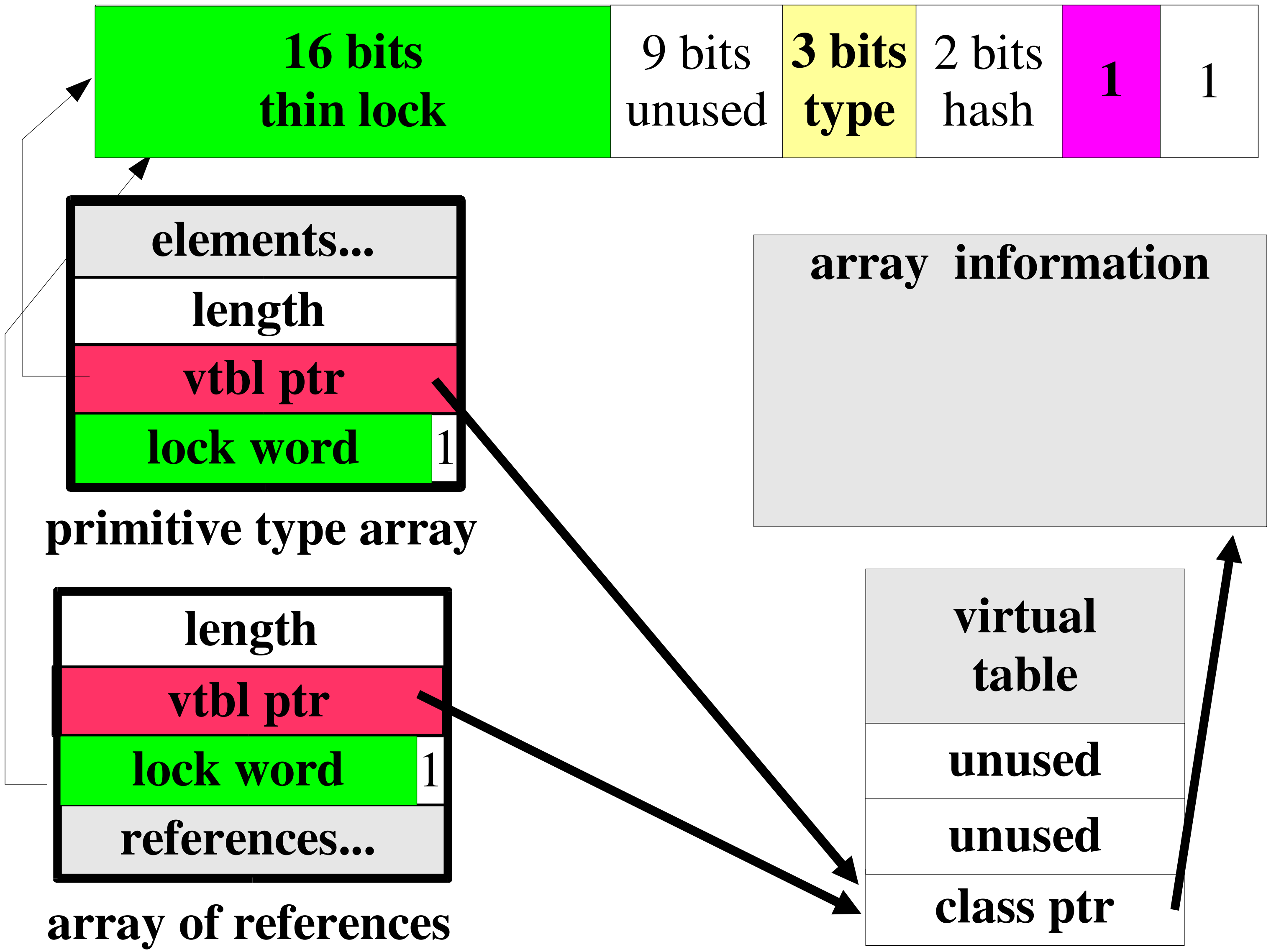
# New Bidirectional Object Layout



# Object Instance Header Layout



# Array Instance Header Layout



# Advantages of Bidirectional Objects

- Tracing most objects can be done without accessing memory than the lock word and reference fields/elements
- Only bigger objects (not arrays) require access to information in the virtual table
(1 indirection, at most 2 words accessed)
- Garbage collection **never** requires access to class information anymore
- Reference fields are placed sequentially, which is usually good for hardware memory caches

# Sparse Interface Virtual Tables

- Java supports multiple inheritance of interfaces

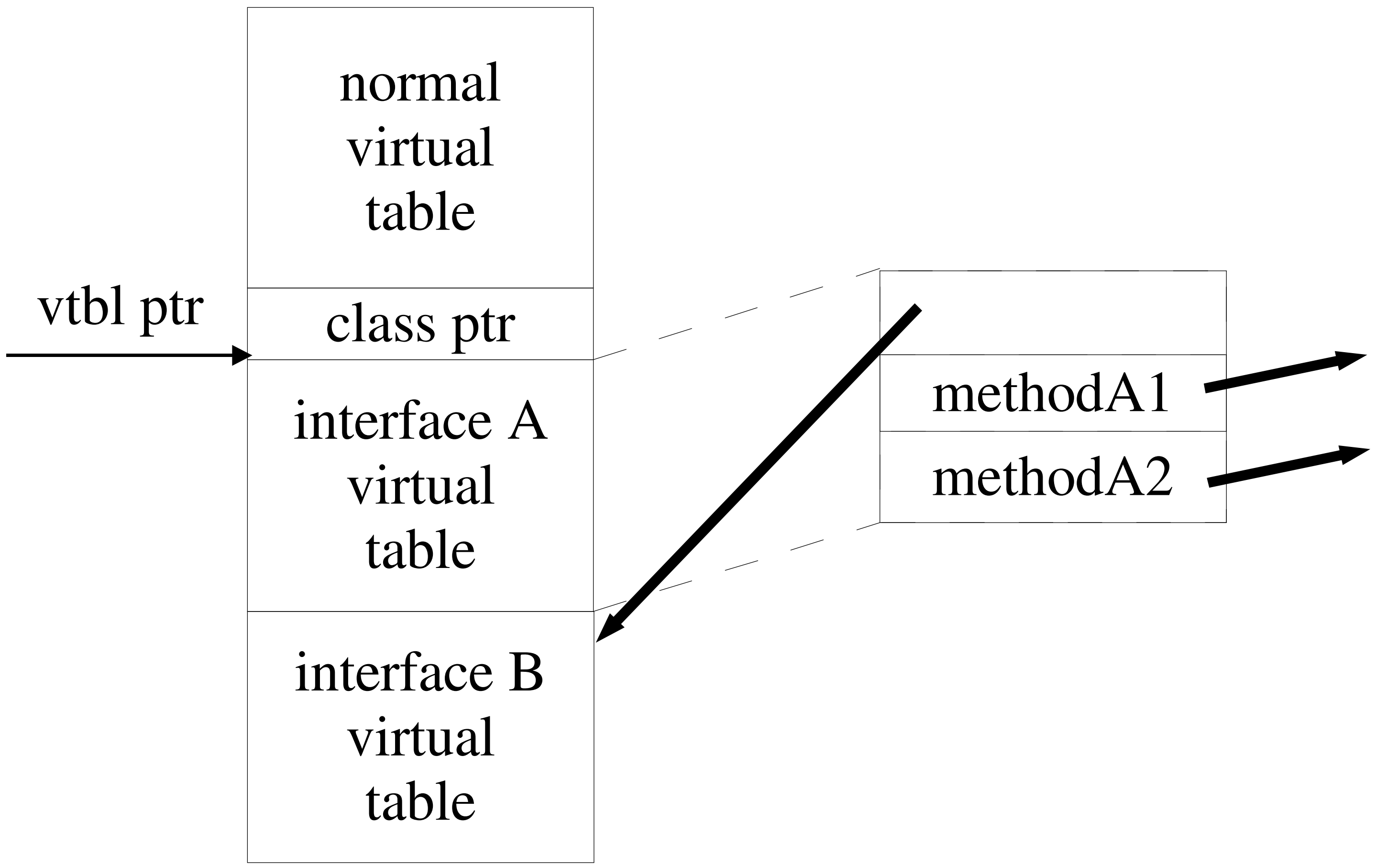
```
interface A {
  void methodA1();
  void methodA2();
}
```

```
interface B {
  void methodB1();
  void methodB2();
}
```

```
interface C
 implements A, B {
...
}
```

- Traditional interface table layout (C++ like) requires variable time (linear) method lookup in presence of dynamic class loading
- We introduce a simple yet effective solution that does not require dynamic code rewrite

# Traditional Layout

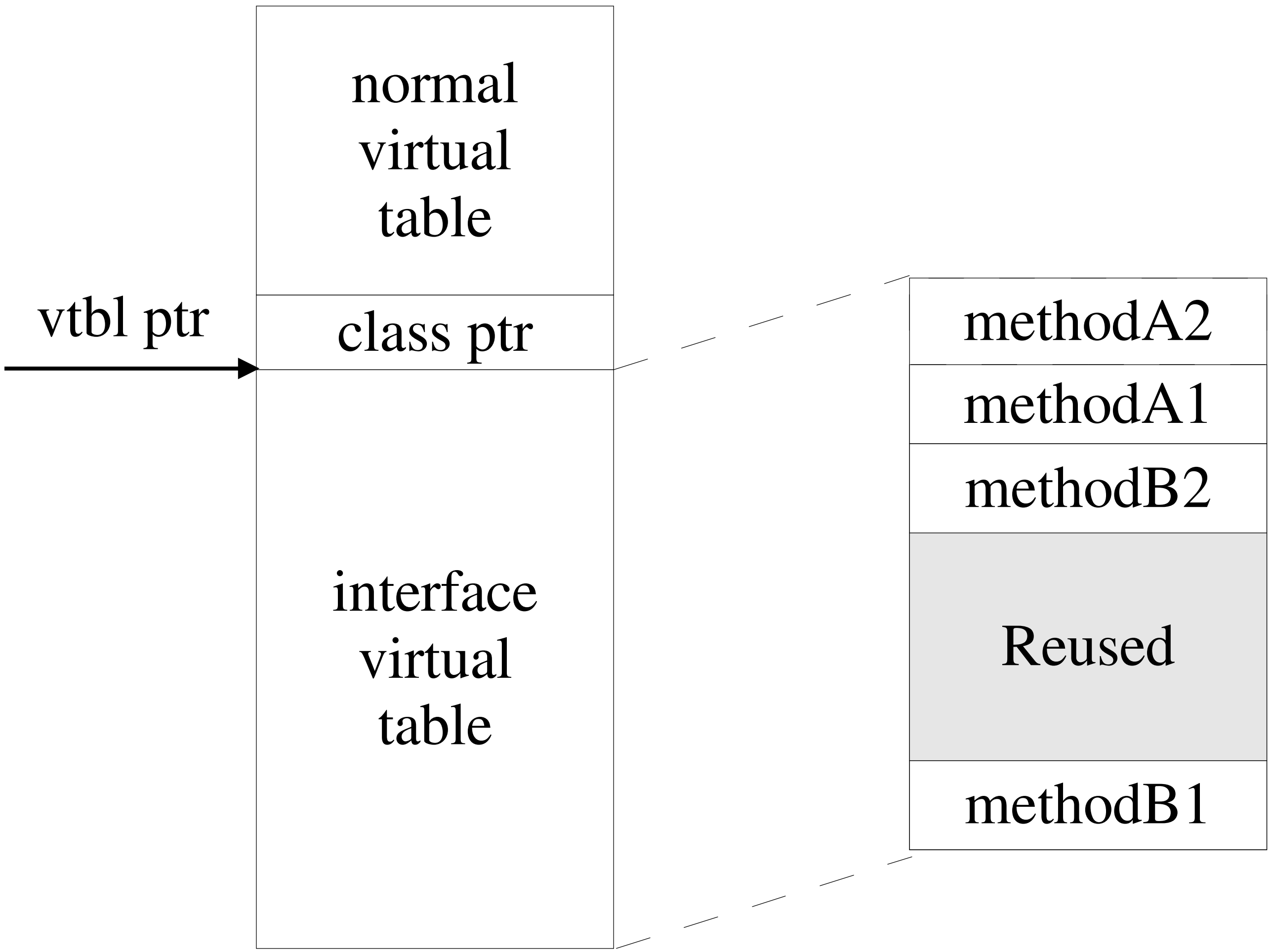



virtual table of a class implementing interfaces A and B

# Sparse Interface Virtual Tables

- Technique:
  - Assign a <u>globally unique</u> decreasing ID (–1, –2,...) to every interface method *<u>signature</u>*
  - We use this ID as an index into a per class unique interface virtual table
  - The length of the interface table is determined by the lowest ID of all methods found in interfaces implemented by the class
- Invoking an interface method has now the same cost as invoking a normal virtual method
- The interface table is sparse: it contains holes. We reuse this memory for storing other class related information (on a class loader basis).

# New Sparse Layout




virtual table of a class implementing interfaces A and B

# Thin Locks

- Bacon [PLDI98]
  - 24 bit lock/object.    0 =  unlocked
  - use `compare-and-swap` to lock [value:Thread ID]
  - If contention: spin lock, then inflate lock
  - Advantages:
    - fast lock (1 atomic op),  fast unlock (without atomic op)
- Onodera [OOPSLA 99]
  - Use a *contention bit*  *(*in another instance word)
  - On contention: set bit, then wait
  - On unlock, check bit

# Busy–wait Free Thin Locks

- Problems in [Onodera]:
  - 1 bit per instance = 1 word per instance (because of alignment in small objects)
- Our solution:
  - Put *contention bit* in thread structure instead
- Setup (per thread structure):
  - 1 contention bit
  - 1 contention lock (MUTEX)
  - 1 linked list of tuples (waiting thread, object)
- On contention: (1) lock MUTEX, (2) set contention bit, (3) add tuple, then (4) wait [simplified!]
- On unlock: check bit and signal waiting threads

# Experimentation

- Benchmarks:
  - Compress: One of the SPEC benchmarks
  - SableCC: A parser generator, works in two main phases: (1) lexer generation and (2) parser generation (throws and catches exceptions)
  - MiniBasic: A small BASIC interpreter, that operates using a visitor object on the abstract tree of the source program.
  - ArrayAlloc: A micro benchmark that allocates 1 million objects in a "Vector" like data structure (growing array)

# Raw Performance

| | Compress | SableCC lexer / all | MiniBasic | ArrayAlloc |
|---|---|---|---|---|
| SableVM | 8m15.1s | 4.4s / 22.9s | 1m16.6s | 5.5s |
| Jdk 1.2.2 interpreter | 9m49.1s | 5.1s / 7.4s | 25.3s | 7.5s |
| Kaffe 1.0.6 interpreter | 70m27.5s | fails | 5m40.0s | 35.4s |
| Jdk 1.2.2 JIT | 1m39.8s | 3.0s / 4.7s | 10.0s | 5.8s |
| Kaffe 1.0.6 JIT | 2m40.5s | fails | 32.4s | 11.0s |
| Kaffe 1.0.6 JIT3 | 1m9.7s | fails | 25.3s | 13.5s |

# Interpreter Versions

| | switch based | direct threaded | inlined threaded |
|---|---|---|---|
| Compress | 14m15.5s | 11m23.7s | 8m15.1s |
| SableCC | 31.3s | 23.8s | 22.9s |
| MiniBasic | 1m20.2s | 1m9.7s | 1m16.6s |
| ArrayAlloc | 6.5s | 6.2s | 5.5s |

# Garbage Collection (precise copying collector)

| | av. survival (bytes) | bidirectional objects | indirect ref. info | bid.obj. saving |
|---|---|---|---|---|
| Compress | 9,868,801 | 1s 388,360μs | 1s 524,843μs | 8.3% |
| SableCC | 704,937 | 158,661μs | 204,174μs | 22.2% |
| MiniBasic | 244,828 | 46,692μs | 59,231μs | 21.2% |
| ArrayAlloc | 7,891,440 | 343,418μs | 485,254μs | 29.2% |

# Sparse Virtual Interface Tables

| | classes with i.v.table | total size (bytes) | holes (bytes) |
|---|---|---|---|
| Compress | 19 | 2,396 | 692 |
| SableCC | 279 | 220,576 | 176,388 |
| MiniBasic | 61 | 18,716 | 11,464 |
| ArrayAlloc | 8 | 564 | 32 |

# Conclusion

- Portable, yet efficient open–source JVM
- Research framework
- Performance enhancement techniques
  - bidirectional object layout
  - sparse interface virtual tables
  - busy–wait free *thin locks*
- Inlined threading is effective for Java bytecode interpreters