VM-based STM

- Is changing the VM acceptable?
- Benefits
  - Direct memory access
  - Full control over garbage collector (GC)
  - Full control over bytecode instructions behavior
  - Can manipulate thread’s header
  - HTM compatible

ByteSTM

- Implicit transaction

```
atomic{
    A = B;
   B++;
}
```

Or:

```
stm.STM.xBegin();
    A = B;
   B++;
stm.STM.xCommit();
```

Implicit transaction

```
Transaction T;
T.begin()
do{
    A.txWrite(B.txRead());
   B.txWrite(B.txRead() + 1);
}while( ! T.commit());
```

Explicit transaction

ByteSTM

- No special transactional instructions
  - Bytecode instructions have two modes
    - Transactional
    - Non-transactional
  - Two new bytecode instructions only
  - One copy of the code
- Works on all data types
  - Memory access is monitored at the bytecode instructions level
- Supports external libraries

ByteSTM

- Atomic blocks anywhere in the code
  - Saves program state at transaction start
  - Restores the saved state when transaction aborted
  - Monitors less objects

```
int  c=10;
c  = a + 5;
atomic{
   c = c / 2;
   a = c;
}
```

```
@Atomic
void  method(int c){
   c = c / 2;
   a = c
}
```

ByteSTM

- Memory model
  - Direct memory access
    - Faster write back
  - Raw memory model
    - One code to handle all cases
    - Moving GC compatible (Absolute address is not used)

Instance field: Object address + field offset
Static field: Static memory address + field offset
Array element: Array address + element size x element index

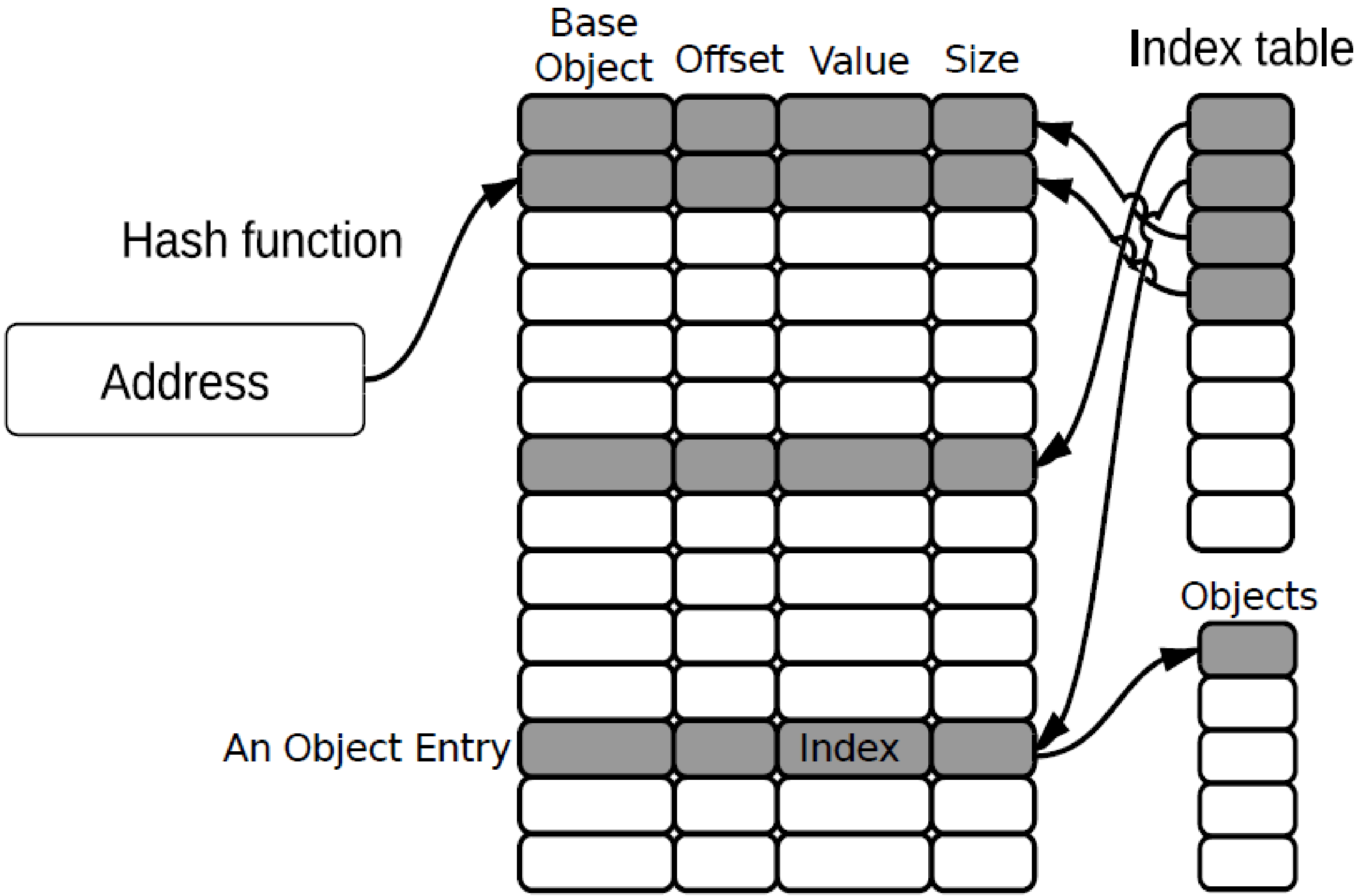
Absolute address

	Data Type	Base Object	offset	Value	Size
Obj1.x	int	Obj1	0	20	4
Obj1.y	double	Obj1	4	46	4
Obj2.obj	Object (reference)	Obj2	0	0 (index)	4

Raw memory model

ByteSTM

- Write-set
  - – Arrays of Primitive + Open Addressing Hashing



ByteSTM

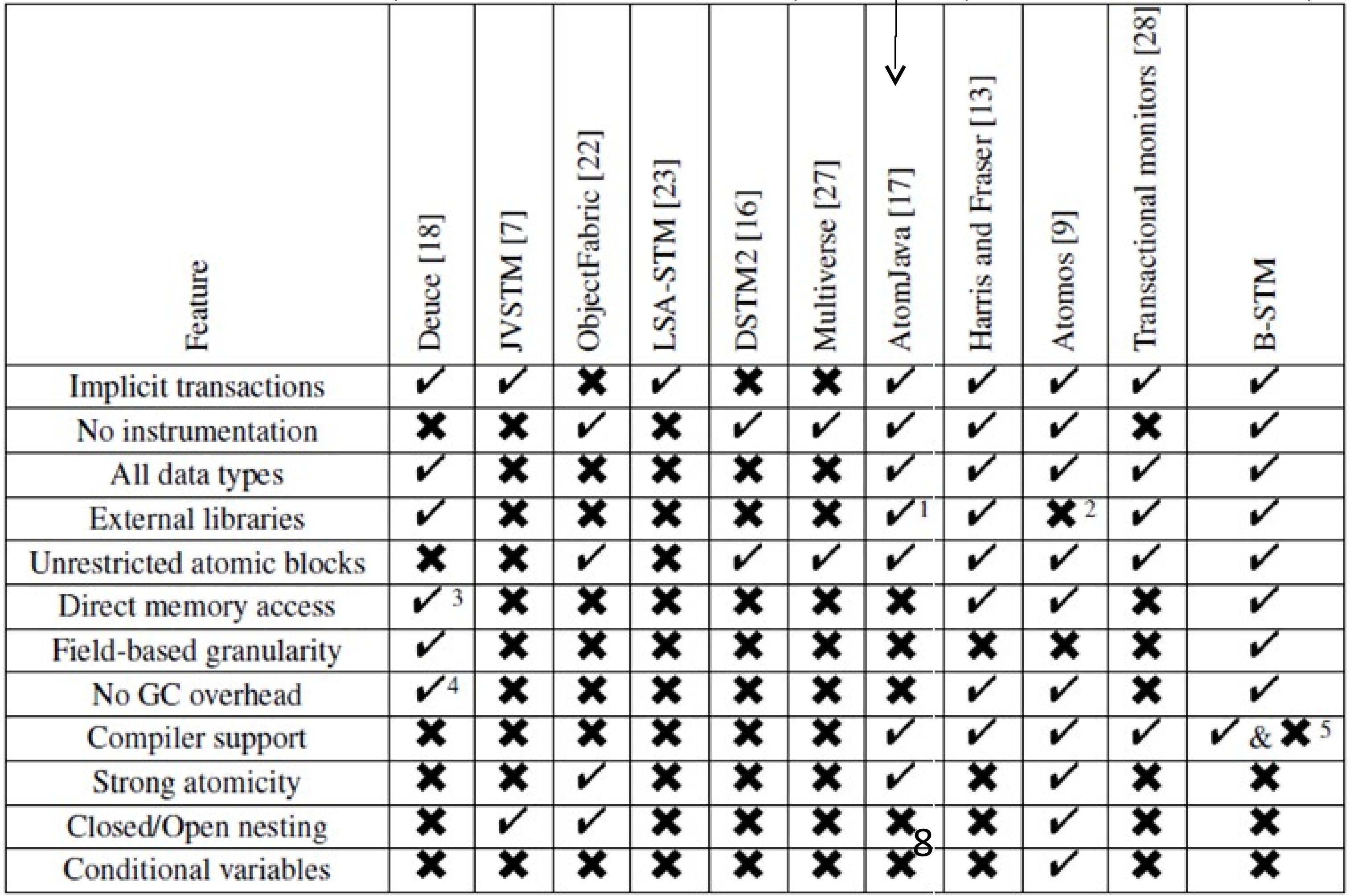
- Metadata in the thread header
  - Faster than Java standard `ThreadLocal`
- No GC overhead
  - Manually allocates and recycles memory for transactional metadata
  - Directly fix write-set only referenced objects

Related Work

Library-based

Compiler-based

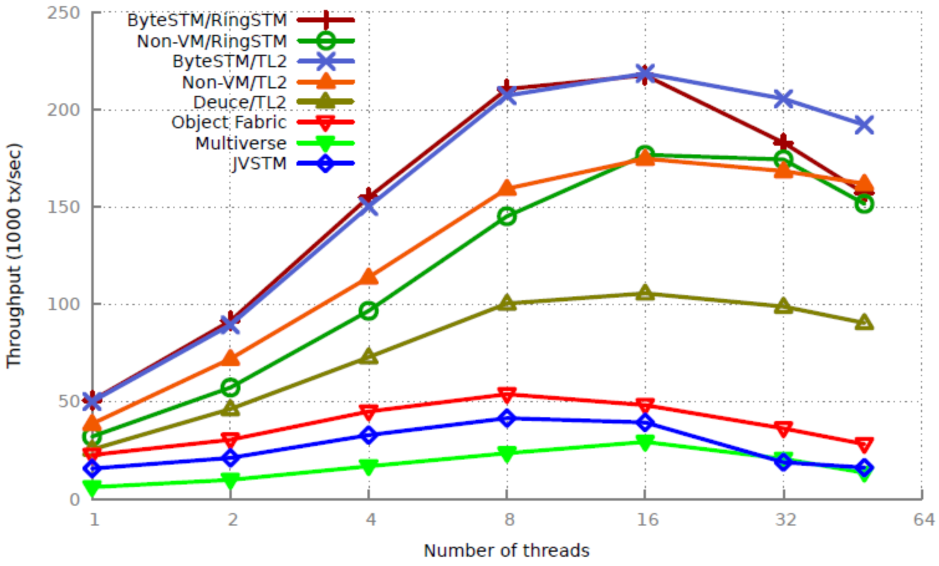
VM-based



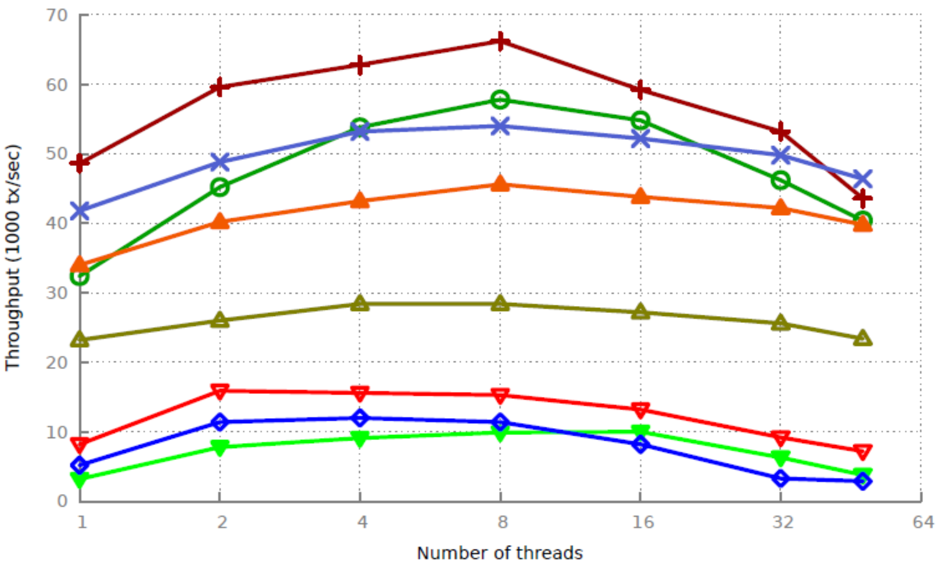
Feature	Deuce [18]	JVSTM [7]	ObjectFabric [22]	LSA-STM [23]	DSTM2 [16]	Multiverse [27]	AtomJava [17]	Harris and Fraser [13]	Atomos [9]	Transactional monitors [28]	B-STM
Implicit transactions	✓	✓	✗	✓	✗	✗	✓	✓	✓	✓	✓
No instrumentation	✗	✗	✓	✗	✓	✓	✓	✓	✓	✗	✓
All data types	✓	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓
External libraries	✓	✗	✗	✗	✗	✗	✓[1]	✓	✗[2]	✓	✓
Unrestricted atomic blocks	✗	✗	✓	✗	✓	✓	✓	✓	✓	✓	✓
Direct memory access	✓[3]	✗	✗	✗	✗	✗	✗	✓	✓	✗	✓
Field-based granularity	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
No GC overhead	✓[4]	✗	✗	✗	✗	✗	✗	✓	✓	✗	✓
Compiler support	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓ & ✗[5]
Strong atomicity	✗	✗	✓	✗	✗	✗	✓	✗	✓	✗	✗
Closed/Open nesting	✗	✓	✓	✗	✗	✗	✗	✗	✓	✗	✗
Conditional variables	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗

Performance

- Linked List



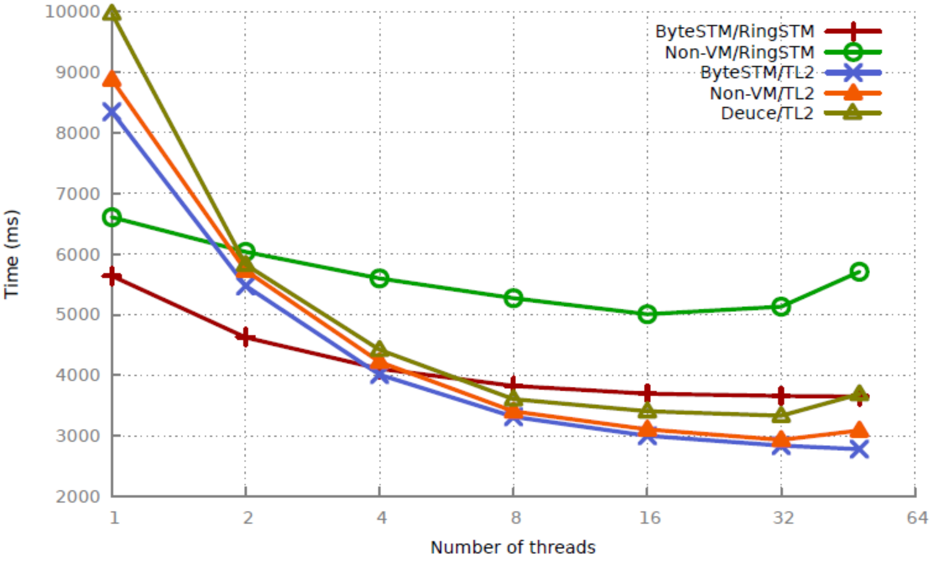
(a) 20% writes.



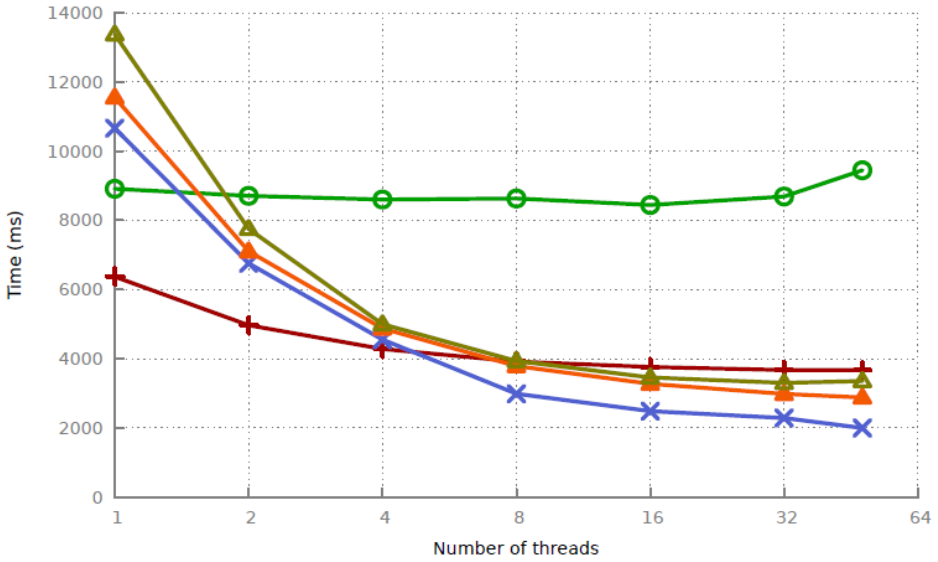
(b) 80% writes.

Performance

- Vacation



(a) Low Contention



(b) High Contention