

Jupiter: A Modular and Extensible JVM

Patrick Doyle and Tarek Abdelrahman

Edward S. Rogers, Sr.
Department of Electrical and Computer Engineering
University of Toronto

{doylep|tsa}@eecg.toronto.edu

Outline

- Motivation.
- Jupiter design.
- Configuration example (object allocator).
 - Flexibility.
 - Performance.
- Status of the implementation.
- Conclusions and future work.

Research into Scalable JVMs

- **Research goal:** to investigate JVM architectures to deliver high performance on large-scale parallel systems.
 - 128-processor cluster of workstations with software-based SVM.
 - Single system image (SSI).
- Research on Java-based SSI on clusters to date is limited to small numbers of processors.
 - cJVM
 - Jessica
 - Hyperion
 - etc.
- It remains unclear how to design a JVM to scale well to large numbers of processors.

Approaches to Scalability

- Four main target areas:
 - Memory locality.
 - Garbage collection.
 - Memory consistency.
 - Support for threading and synchronization.
- We will investigate alternative approaches in each area.

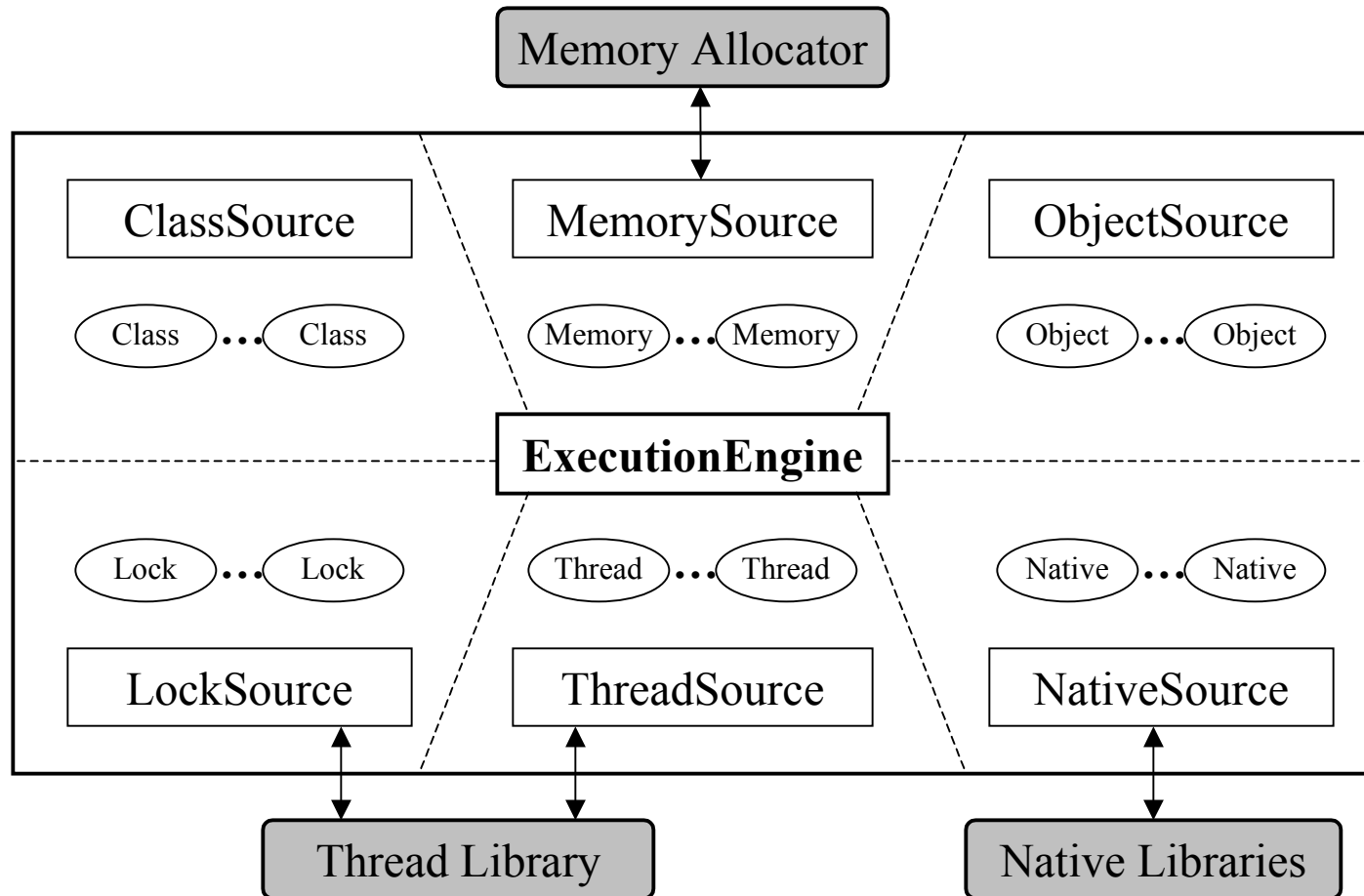
JVM Infrastructures

- In order to carry out this research, we need a JVM infrastructure that is:
 - Modular.
 - Flexible/Extensible.
 - Efficient.
- Currently available JVMs:
 - Hard to modify (Kaffe or Sun JVM).
 - Designed to address a specific aspect of JVM performance.
- Hence, we elected to build our own infrastructure.
 - Hard.
 - Rewarding.

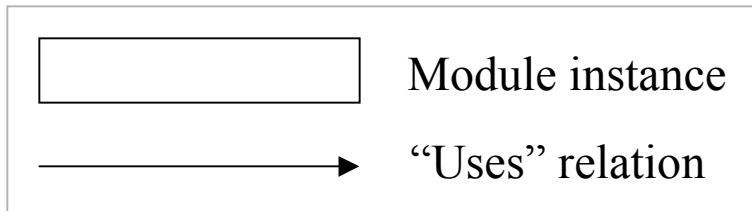
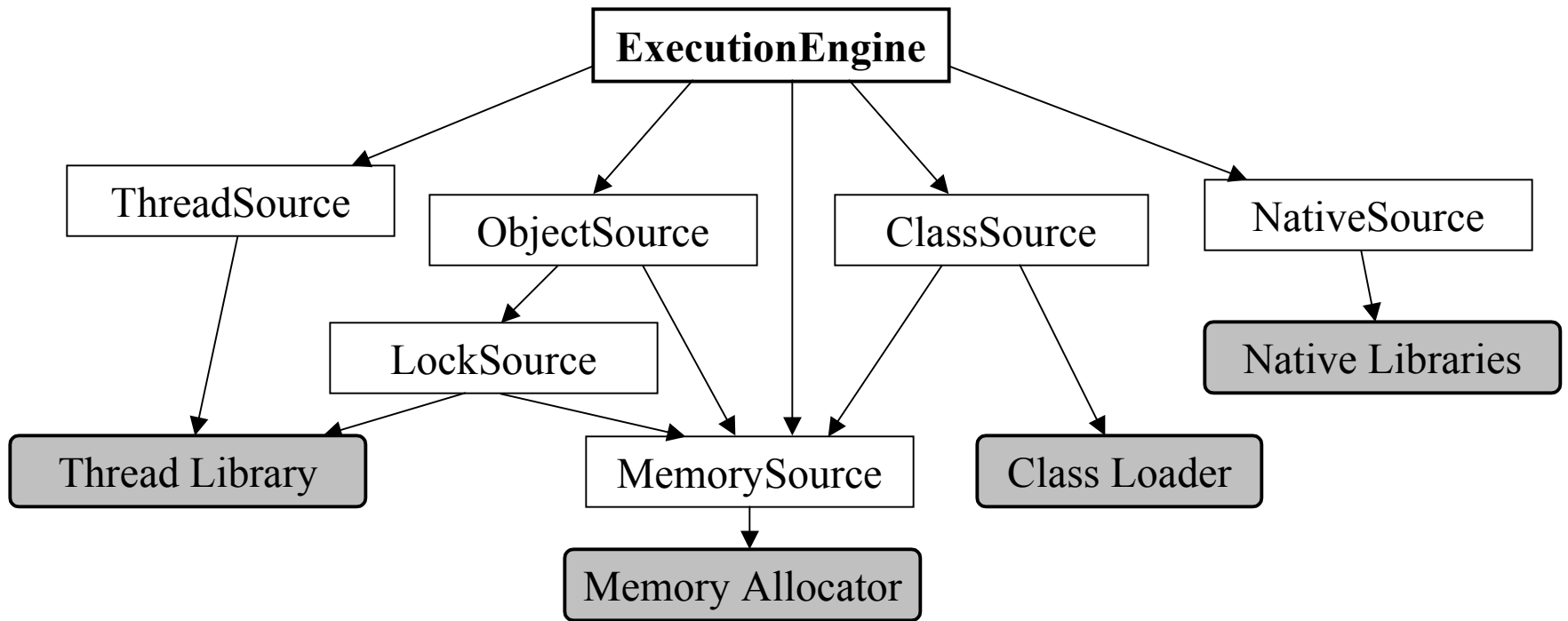
Jupiter Philosophy

- Jupiter is assembled out of small modules.
 - Much like UNIX pipelines.
 - Interconnection of modules determines JVM behavior.
- Module requirements:
 - **Atomicity**: modifications should not split modules.
 - **Cohesion**: modifications should involve few modules.
 - **Independence**: unrelated modifications should be orthogonal.
 - Must find a balance.
- Two kinds of modules
 - **Facilities**: manage resources.
 - **Resources**: used by a running Java program (e.g. Classes, Objects, Threads, etc.).

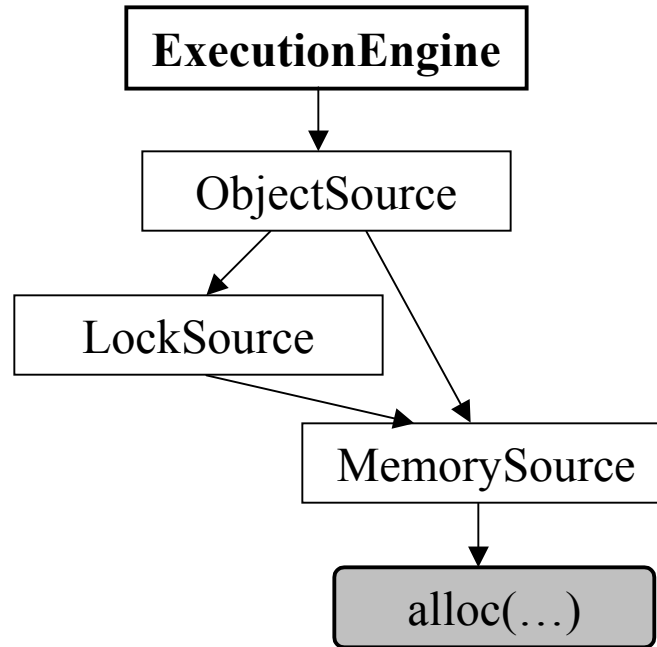
Jupiter Overview



An Incarnation of Jupiter



Object Allocator Example

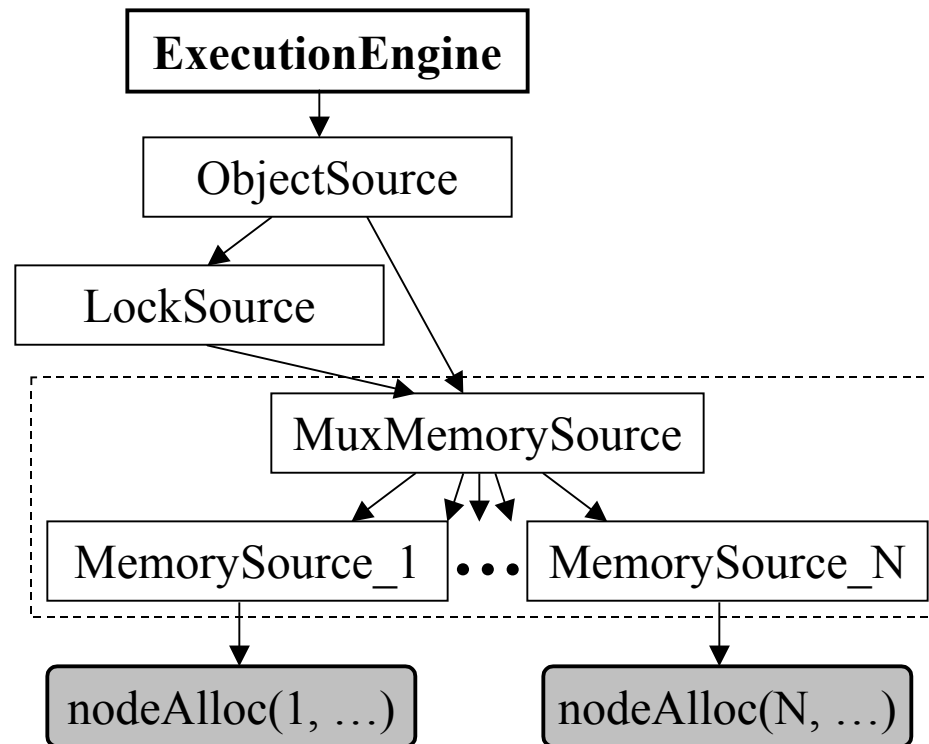


- Extend to achieve locality:

```
void *nodeAlloc(int nodeNumber, int size);
```

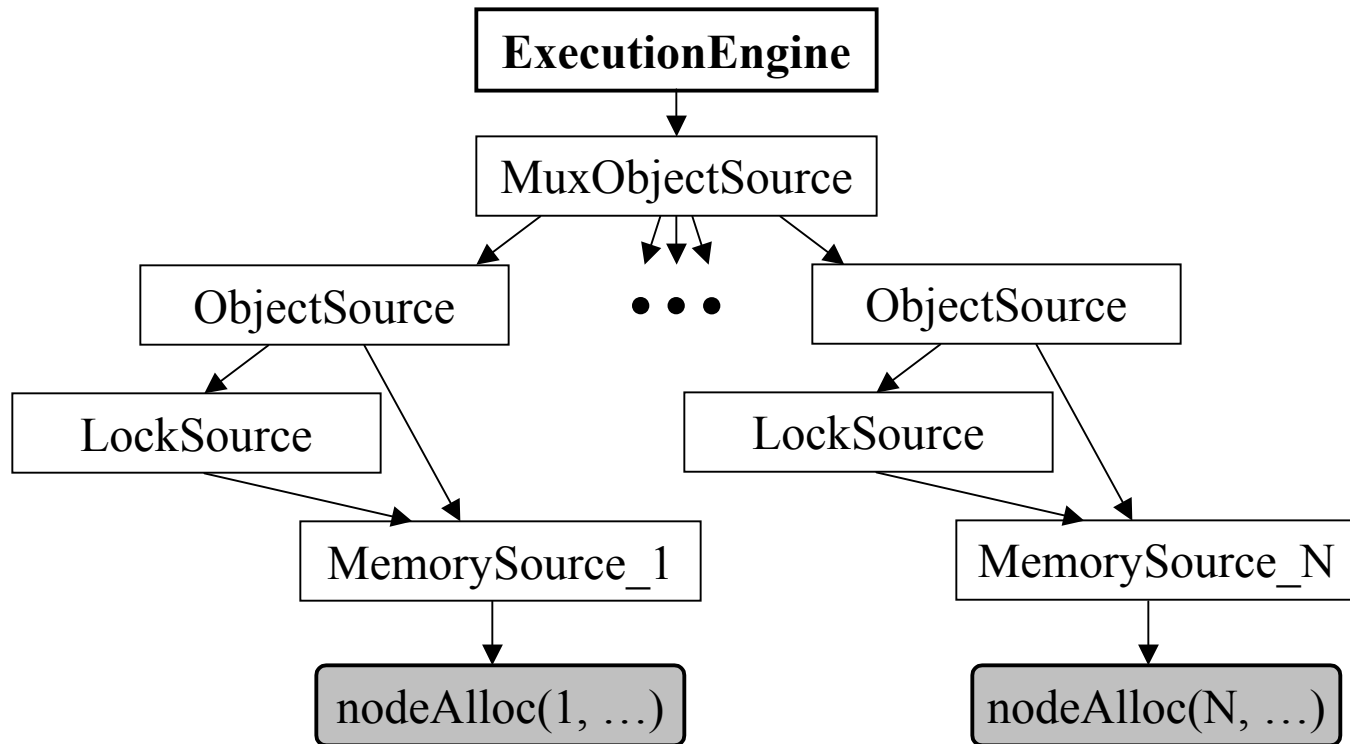
- Multiple levels:
 - Memory allocation level.
 - Object allocation level.
 - Execution engine level.

Node-specific MemorySources



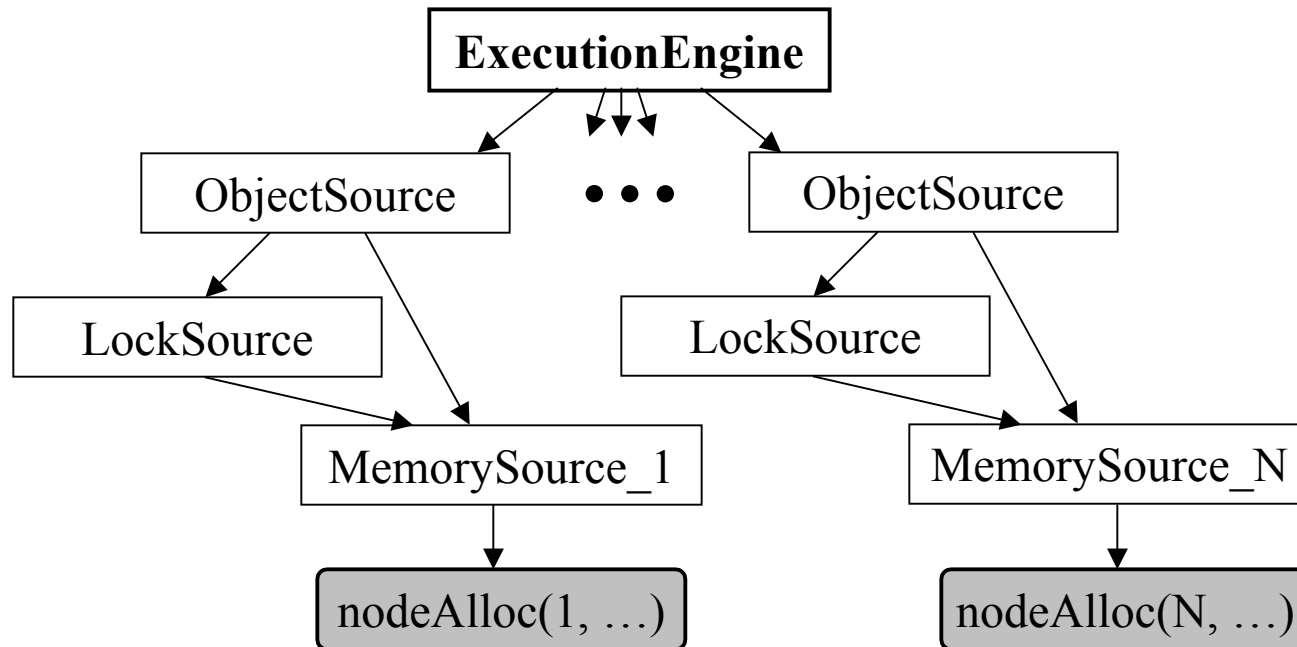
- MemorySource for each node.
 - Same MemorySource interface.
 - Minimal change to the implementation of MemorySource.
 - Locality decisions made transparently in MuxMemorySource.

Node-Specific ObjectSources



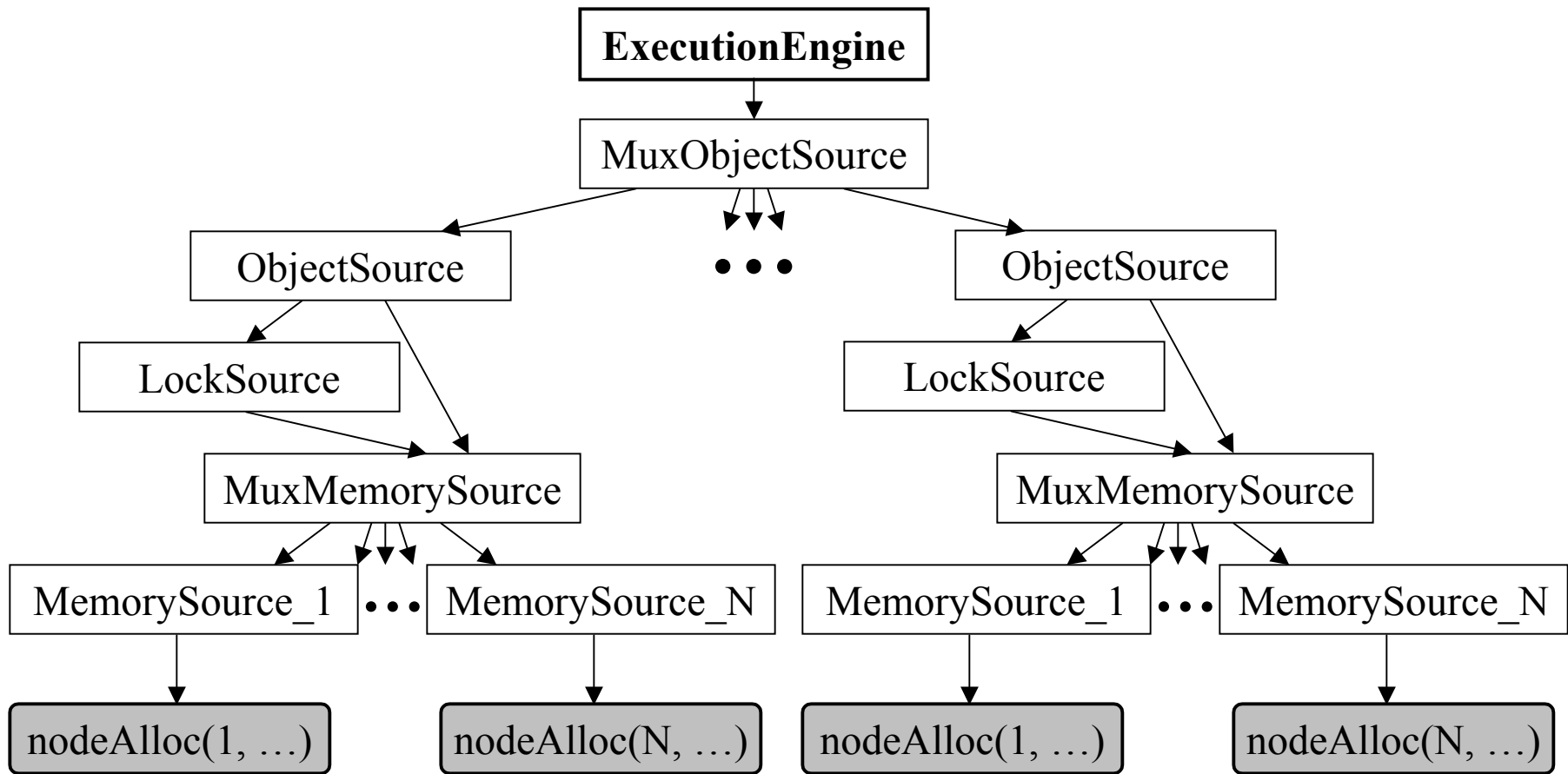
- ObjectSource for each node.
 - Enhances locality without altering ObjectSource interface.
 - Node choices still transparent to **ExecutionEngine**.
 - Same ObjectSource, LockSource, MemorySource_{*i*} as before.

A Locality-Aware ExecutionEngine



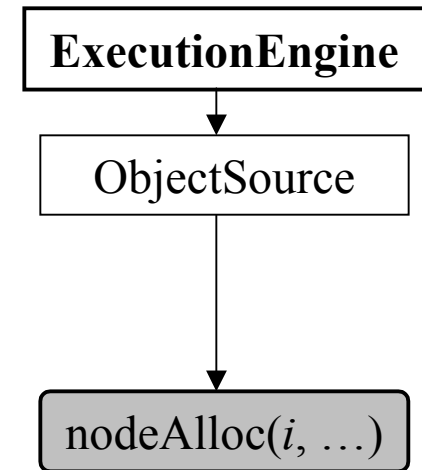
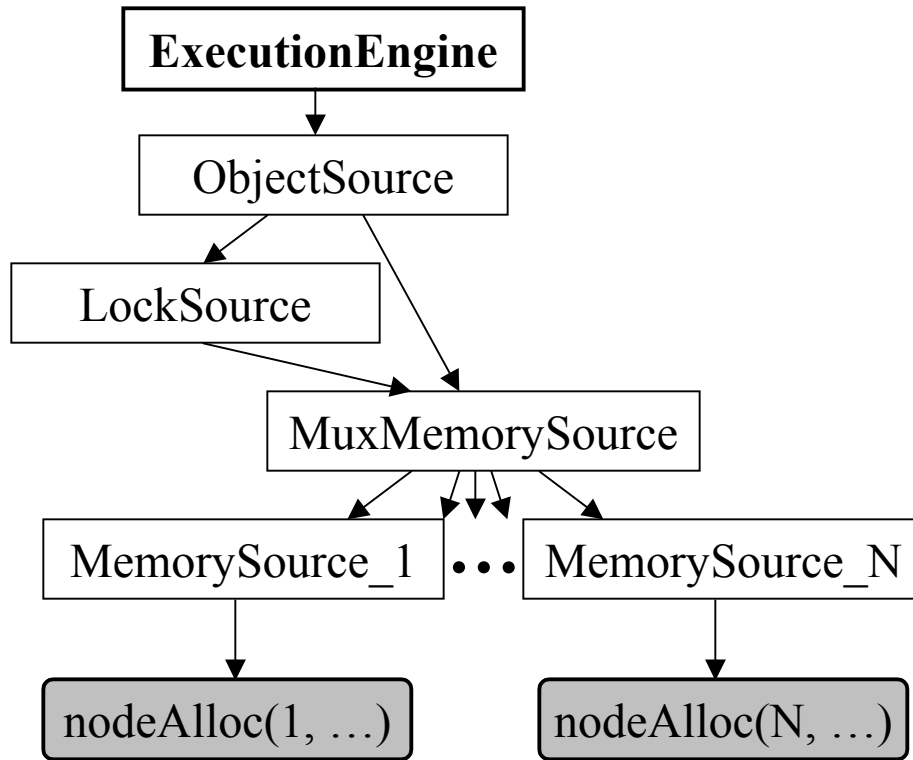
- **Locality-aware ExecutionEngine.**
 - **ExecutionEngine** directly manages locality.
 - Same **ObjectSource**, **LockSource**, **MemorySource_i** as before.
 - Modifications confined to **ExecutionEngine**.

Combined Approach



- Modifications confined to a small number of modules.
- Multiple configurations through module interconnections.

Performance



- Two problems:
 - **Call overhead**: must make calls through the module hierarchy.
 - **Object proliferation**: **MemorySource_i** for each node *i*.

Performance

- Standard MemorySource declarations:

```
typedef struct ms_struct *MemorySource;  
void *ms_getMemory(MemorySource this, int size);
```

- MemorySource constructors:

```
MemorySource ms_new(int nodeNumber);  
MemorySource ms_newMux();
```

- MemorySource usage:

```
void *ptr = ms_getMemory(obs_memorySource(), size);
```

Performance

- Custom declarations for nodeAlloc:

```
typedef int MemorySource;
```

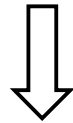
```
inline MemorySource ms_new(int nodeNumber) {  
    return nodeNumber;  
}
```

```
inline MemorySource ms_newMux() { return -1; }
```

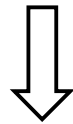
```
inline void *ms_getMemory(MemorySource this, int size) {  
    if(this == ms_newMux())  
        return nodeAlloc(/* Some node */, size);  
    else  
        return nodeAlloc(this, size);  
}
```

Performance

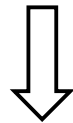
```
void *ptr = ms_getMemory(obs_memorySource(), size);
```



```
void *ptr = ms_getMemory(ms_newMux(), size);
```

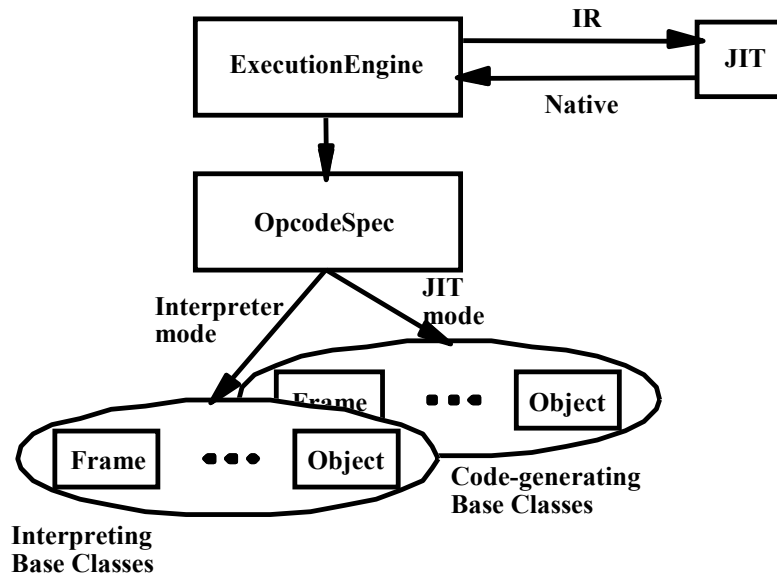


```
void *ptr = ms_getMemory(-1, size);
```



```
void *ptr = nodeAlloc(/* Some node */, size);
```

JIT Compiler



- Use OpcodeSpec module that is always executed by the **ExecutionEngine**.
- OpcodeSpec functions:
 - Interpret bytecode, or
 - Emit IR, which is passed to the JIT compiler.

Status

- Project started in March 2001.
- Core JVM implemented in C:
 - All single-threaded facilities and resources.
 - Interpreter-based execution engine.
 - Native interface: java-to-native calls, but no callbacks.
- Can execute simple programs.
- Performance poor compared to Kaffe and Sun, mostly due to implementation problems.

Conclusions and Future Work

- We presented a JVM structure designed to be modular, flexible and efficient.
 - Instances of well-defined modules.
 - Interconnection of modules.
 - Performance.
- The project is in its early stages, and future work will focus on completing the implementation.
- Research goals.