Facade: A Compiler and Runtime for (Almost) Object-Bounded Big Data Applications

Khanh Nguyen, Kai Wang, Yingyi Bu, Lu Fang, Jianfei Hu, Harry Xu

University of California, Irvine

Motivation

Design a scalable and efficient system is a key challenge to both researchers and practitioners

- Mainstream approach is to enable parallelism by using a large number of machines
- Typical parallel frameworks such as MapReduce, Giraph, Hive, or Pig use Java, a managed language which comes with a managed runtime system

When object-orientation meets Big Data, the cost of managed runtime system becomes the bottleneck

- *Significantly reduced scalability*: JVM crashes even if the size of the processing dataset is much smaller than the heap size
- *Prohibitively high memory management cost*: GC time accounts for up to 50% of the overall execution time [Bu ISMM'13]

Poor performance is inherent with the managed runtime system and remains a serious problem despite many optimizations from various research communities

Related Work

- Optimizations of Big Data applications:
  - Data pipeline: [Agrawal VLDB'08], Flume-Java [Chambers PLDI'10], DryadLINQ [Yu OSDI'08]
  - MapReduce-related: Hive [Thusoo ICDE'10], Panacea [Liu CGO'10]
- Techniques for reducing runtime management costs
  - [Aiken PLDI'95], [Hallenberg PLDI'02], [Hick ISMM'04]
  - Immix [Blackburn PLDI'08]
  - Prolific types [Shuf POPL'02]
- Techniques for reducing numbers of objects
  - Object pooling and certain design patterns
  - Object inlining: [Dolby PLDI'00]
  - Pool-based allocation: [Lattner PLDI'05, PLDI'07]

Golden Rule for Scalability

- The number of heap objects and references must not grow proportionally with the cardinality of the dataset
- Formally, Facade guarantees a static bound:

O(s)
- s : cardinality of the dataset

O(t*n+p)
- t : number of threads
- n : number of data types
- p : number of pages

- Although t and p cannot be bounded statically, they are usually very small, hence the total number of objects is "almost" statically bounded
- The reduction is in many orders of magnitudes: in PageRank (GraphChi) 14 billions data objects are reduced to 1363 objects

Facade Programming Model

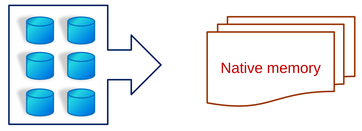
- Breaking the long-held object-oriented programming principle:
  "Objects are used both to store data and to provide data manipulation interfaces"
- Facade makes a clear separation between data representation and data manipulation

Benefits from Facade

- Smaller memory consumption
- Significantly reduced GC time
- Reduced execution time
- Improved scalability

Data Representation

- Data objects are stored in native memory. A memory page is a fixed-length contiguous block of memory, obtained through the `Unsafe` interface
- The layout of an object in a memory page is exactly the same as the way the object was stored in the heap
- Each data object becomes a data record and uses absolute memory address as its reference (pointer)




```
class Professor {
    int id;
    Student[] students;
    String name;
}

class Student {
    int id;
    String name;
}
```

Record type	Address	Type	Lock	id	Fields	name
Professor	0x04e0	12	0	1254	0x0504	0x070a
Student[]	0x0504	25	253	9	0x0800	…
String	0x070a	4	…	…		
		…				students
Student	0x0800	13	0	2541	0x0868	

Iteration-based Memory Management

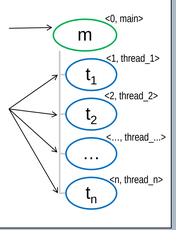
- *Iteration definition*
  - Iterations are easy to identify; Facade relies on a user-provided pair of calls to start/end iterations
  - Nested iterations are supported
- *Allocation*: customized, high-performance allocator
  - Create a page manager per thread and per iteration to control memory; page managers form a hierarchy
  - Each page manager is associated with a pair <*iterationID, thread*>
  - Contiguous allocations get contiguous space to maximize data locality
  - Support mostly thread-local data allocation
- *Deallocation*:
  - Once an iteration ends, reclamation can be safely done concurrently on the sub-tree rooted at the current page manager

```
iteration_start()
  …
 thread[i].run() {
    iteration_start()
 }
  …
iteration_end()
```

Iteration starts: create and register a manager m which is in main thread

Sub-iteration starts: create and register manager t_i as child of manager m
Note that it is not required to signal the end of the sub-iteration

Iteration ends: memory reclamation is started on the sub-tree rooted at m



Data Manipulation

- Heap objects are created as facades and used only for control purposes: resolve method calls, perform dynamic type check, pass parameters, return values,…

D (user-specified data class) —automatic→ DF (Facade class)

- Facade class DF
  - Has only one instance field pageRef
  - Does NOT contain actual data
  - Contains all methods in data class D
  - Its instances are controlled by bounded object pooling
- *Bounded facade pooling*
  - A new approach to statically bound the number of DF's instances
  - Generate, for each data type, a pool whose size is the maximum number of operands of that type required by an instruction
  - At any program point, every element of a pool is available for use

Transformation Example

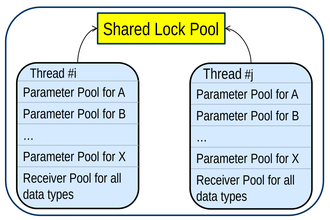
Original code

```
class Professor {
 int id;
 Student[] students;
 String name;


void addStudent(Student s,
                int i) {



 students[i] = s;

}
}


static void client
            (Professor f) {

 Student s = new Student();









 Professor p = f;
 Student t = s;


 p.addStudent(t,0);









}
```

Facade-generated code

```
class Facade { long pageRef; …}

class ProfessorFacade extends Facade {
 static int id_OFFSET = 0;
 static int students_OFFSET = 4;
 static int name_OFFSET = 8;

/* modified method's signature */
void addStudent(StudentFacade sf, int i) {
 /* release facades' binding */
 long thisRef = this.pageRef;
 long sRef = sf.pageRef;

 FacadeRuntime.writeArray(
    thisRef, students_OFFSET, i, sRef);
}
}


static void client(ProfessorFacade pf) {
 long fRef = pf.pageRef;

 long sRef = FacadeRuntime.allocate
    (Student_TypeID, Student_RecordSize);
 StudentFacade sf = studentPool[0];
 sf.pageRef = sRef;
 sf.facade$init();

 long pRef = fRef;
 long tRef = sRef;

 /* retrieve facades from pool */
 ProfessorFacade pf2 = professorPool[0];
 StudentFacade sf2 = studentPool[0];
 /* bind facades with references */
 pf2.pageRef = pRef;
 sf2.pageRef = tRef;
 pf2.addStudent(sf2,0);

}
```

Concurrency Support



A, B,…, and X are data types

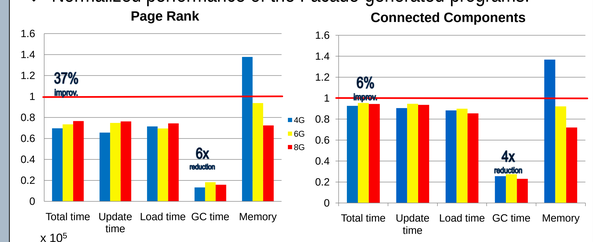
- Thread-local pooling for facade objects
- One shared lock pool
  - Each object is an instance of a special lock class
  - The number of locks simultaneously needed is bounded by the number of threads

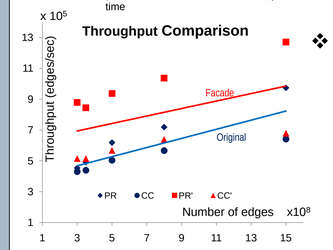
Implementation and Evaluation

- Implemented using the Soot framework [Vallée-Rai CC'00]
- Most of the Java 7 features are supported
- 40+ KLOC; 1.5 years of development
- Fast translation speed: 950 instructions per second on average

GraphChi [Kyrola OSDI'12]

- A high-performance graph analytical framework that enables efficient processing of large graphs on a single machine
- Normalized performance of the Facade-generated programs:

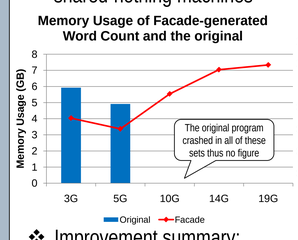


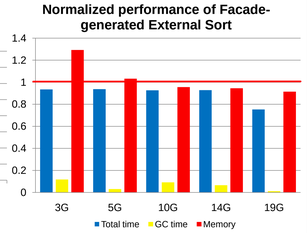


- Improvement summary:
  - ✓ 5x reduction in GC time
  - ✓ 7%-28% reduction in memory consumption (6+GB datasets)
  - ✓ 17% reduction in running time (avg.) max 48%
  - ✓ 1.4x improvement in throughput

Hyracks [Borkar ICDE'11]

- A data parallel platform to run data-intensive jobs on a cluster of shared-nothing machines





- Improvement summary:
  - ✓ 25x reduction in GC time, max 88x
  - ✓ 7% reduction in memory usage (avg.), max 32%
  - ✓ 10% reduction in running time (avg.), max 25% (External Sort)
  - ✓ Scalability is significantly increased (3.8x Word Count)

GPS [Salihoglu SSDBM'13]

- A distributed graph processing system developed for scalable processing of large graphs
- Improvements on Page Rank, KMeans and Random Walk:
  - ✓ 3-15% running time reduction
  - ✓ 10-40% reduction in GC time
  - ✓ 14% space reduction

Conclusions

- A complete, non-intrusive package with a compiler that can automatically transform existing programs and a runtime system that runs on top of a JVM
- Experimental results show significant improvement in execution time, memory consumption, and scalability