Is It Possible to Code an Efficient Software Switch?

Linearizing the Heap, and the Pervasive Use of Hardware Accelerators

Nick Mitchell
along with Ioana Baldini, Peter Sweeney
IBM T.J. Watson Research Center

July 31, 2014
What Nick Does:
Work with IBM customers and developers to make their applications run well

the data herein is backed by thousands of real apps
Summer School Means
Nick Learns At Least As Much From You, As You From Him
0. The Apps
Most of Our Data Comes from Somewhere Else

Request: MongoDB, Hadoop, FPGA, GPU

Response: My Service, Your Service
Each Source Has its Own Wire Protocol
Each Service Has its Operational Form
Solutions Denied:
(but worth learning from)

keep everything in memory
code in a systems language
Summary

• Programmers are *plumbers* who write *software switches*

• ECOOP Languages offer poor support for *efficient* software switches, because of the distance between *operational* and *wire* forms

• Amdahl’s Law strikes again, severely limiting the use of *specialized computational circuitry*

• What are our options for doing better?
1. Initial Thought Exercises
Implement an Efficient Map

- a conventional chained hash map
  - ~3 pointers per entry
  - ~3-5 cache misses per GET
  - copying required for RPC

- c.f.
Implement This Without Copying Any Bits Into the Heap

update this value
Implement an Efficient String

4–8 bytes of data
24–50 bytes of non-data
8–33% efficiency
2. Programmers as Plumbers
@POST
@Path("cart/add")
@Consumes(MediaType.MULTIPART_FORM_DATA)
@Produces(MediaType.APPLICATION_JSON)

Response addToCart(@CookieParam("session") Cookie session,
                   @FormParam("item") int itemCode,
                   @FormParam("quantity") int quantity) {
    ShoppingCart cart = new ShoppingCart(session);
    cart.add(itemCode, quantity);
    return Response.ok(cart.status())
        .cookie(cart.toCookie())
        .build();
}
POST
@Path("cart/add")
@Consumes(MediaType.MULTIPART_FORM_DATA)
@Produces(MediaType.APPLICATION_JSON)

Response addToCart(@CookieParam("session") Cookie session,
                    @FormParam("item") int itemCode,
                    @FormParam("quantity") int quantity) {

    ShoppingCart cart = new ShoppingCart(session);
    cart.add(itemCode, quantity);
    return Response.ok(cart.status())
                   .cookie(cart.toCookie())
                   .build();
}
Protocol Unwrapping

@POST
@Path("cart/add")
@Consumes(MediaType.MULTIPART_FORM_DATA)
@Produce(MediaType.APPLICATION_JSON)

Response addToCart(@CookieParam("session") Cookie session,
                    @FormParam("item") int itemCode,
                    @FormParam("quantity") int quantity) {

    ShoppingCart cart = new ShoppingCart(session);
    cart.add(itemCode, quantity);
    return Response.ok(cart.status())
                .cookie(cart.toCookie())
                .build();
}
Deserialization into Application Data Structures

```java
@POST
@Path("cart/add")
@Consumes(MediaType.MULTIPART_FORM_DATA)
@Produces(MediaType.APPLICATION_JSON)

Response addToCart(@CookieParam("session") Cookie session,
                   @FormParam("item") int itemCode,
                   @FormParam("quantity") int quantity) {

    ShoppingCart cart = new ShoppingCart(session);
    cart.add(itemCode, quantity);
    return Response.ok(cart.status())
                   .cookie(cart.toCookie())
                   .build();
}
```
Response addToCart(@CookieParam("session") Cookie session,
    @FormParam("item") int itemCode,
    @FormParam("quantity") int quantity) {

    ShoppingCart cart = new ShoppingCart(session);
    cart.add(itemCode, quantity);
    return Response.ok(cart.status())
        .cookie(cart.toCookie())
        .build();
}
Serialize Application Data Structures Back into a Response

@POST
@Path("cart/add")
@Consumes(MediaType.MULTIPART_FORM_DATA)
@Produces(MediaType.APPLICATION_JSON)

Response addToCart(@CookieParam("session") Cookie session,
                    @FormParam("item") int itemCode,
                    @FormParam("quantity") int quantity) {

    ShoppingCart cart = new ShoppingCart(session);
    cart.add(itemCode, quantity);
    return Response.ok(cart.status())
        .cookie(cart.toCookie())
        .build();
}
Marshalling and Data Formats

operational form

serialize

transfer

deserialize

wire form
3. The Profitability Threshold
Trade-offs

• At what point does externalizing a computation become more pain than it’s worth?

• granularity of kernel
• cost of externalization
• accelerator speedup

Amdahl’s Law
Amdahl's Law

make kernel 6.25% faster

Externalization More Expensive

free externalization, kernel is 100% of overall computation
Amdahl’s Law

- Make kernel 6.25% faster

Externalization equivalent to 1% of overall computation, kernel is 100% of overall computation

<table>
<thead>
<tr>
<th>Percentage of Kernel</th>
<th>Externalization Cost</th>
<th>Overall Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.29%</td>
<td>-1.56%</td>
<td>-8.85%</td>
</tr>
<tr>
<td>0.56%</td>
<td>-1.26%</td>
<td>-12.60%</td>
</tr>
<tr>
<td>0.87%</td>
<td>-1.73%</td>
<td>-19.63%</td>
</tr>
<tr>
<td>1.17%</td>
<td>-2.32%</td>
<td>-22.73%</td>
</tr>
<tr>
<td>1.46%</td>
<td>-3.10%</td>
<td>-32.32%</td>
</tr>
<tr>
<td>1.76%</td>
<td>-4.06%</td>
<td>-34.86%</td>
</tr>
<tr>
<td>2.06%</td>
<td>-5.16%</td>
<td>-36.64%</td>
</tr>
<tr>
<td>2.36%</td>
<td>-6.44%</td>
<td>-38.42%</td>
</tr>
<tr>
<td>2.67%</td>
<td>-7.89%</td>
<td>-40.26%</td>
</tr>
<tr>
<td>2.97%</td>
<td>-9.52%</td>
<td>-42.00%</td>
</tr>
<tr>
<td>3.28%</td>
<td>-11.41%</td>
<td>-43.83%</td>
</tr>
<tr>
<td>3.59%</td>
<td>-13.51%</td>
<td>-45.60%</td>
</tr>
<tr>
<td>3.90%</td>
<td>-15.77%</td>
<td>-47.43%</td>
</tr>
<tr>
<td>4.21%</td>
<td>-18.28%</td>
<td>-49.27%</td>
</tr>
<tr>
<td>4.52%</td>
<td>-20.10%</td>
<td>-51.10%</td>
</tr>
<tr>
<td>4.84%</td>
<td>-22.12%</td>
<td>-52.93%</td>
</tr>
<tr>
<td>5.16%</td>
<td>-24.30%</td>
<td>-54.76%</td>
</tr>
<tr>
<td>5.48%</td>
<td>-26.64%</td>
<td>-56.60%</td>
</tr>
<tr>
<td>5.80%</td>
<td>-29.12%</td>
<td>-58.43%</td>
</tr>
<tr>
<td>6.13%</td>
<td>-31.80%</td>
<td>-60.26%</td>
</tr>
</tbody>
</table>

Externalization More Expensive
Amdahl’s Law

make kernel 12.5% faster

<table>
<thead>
<tr>
<th>Percentage</th>
<th>Performance Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.56%</td>
<td>-4.26%</td>
</tr>
<tr>
<td>1.12%</td>
<td>-3.74%</td>
</tr>
<tr>
<td>1.69%</td>
<td>-3.23%</td>
</tr>
<tr>
<td>2.27%</td>
<td>-2.70%</td>
</tr>
<tr>
<td>2.86%</td>
<td>-2.17%</td>
</tr>
<tr>
<td>3.45%</td>
<td>-1.84%</td>
</tr>
<tr>
<td>4.05%</td>
<td>-1.10%</td>
</tr>
<tr>
<td>4.65%</td>
<td>-0.58%</td>
</tr>
<tr>
<td>5.26%</td>
<td>0.00%</td>
</tr>
<tr>
<td>5.88%</td>
<td>0.56%</td>
</tr>
<tr>
<td>6.51%</td>
<td>1.12%</td>
</tr>
<tr>
<td>7.14%</td>
<td>1.69%</td>
</tr>
<tr>
<td>7.76%</td>
<td>2.27%</td>
</tr>
<tr>
<td>8.43%</td>
<td>2.86%</td>
</tr>
<tr>
<td>9.09%</td>
<td>3.45%</td>
</tr>
<tr>
<td>9.76%</td>
<td>4.05%</td>
</tr>
<tr>
<td>10.43%</td>
<td>4.65%</td>
</tr>
<tr>
<td>11.11%</td>
<td>5.26%</td>
</tr>
<tr>
<td>11.80%</td>
<td>5.88%</td>
</tr>
<tr>
<td>12.50%</td>
<td>6.51%</td>
</tr>
</tbody>
</table>
Amdahl’s Law

make kernel 25% faster

<table>
<thead>
<tr>
<th>Percentage</th>
<th>1.01%</th>
<th>2.04%</th>
<th>3.09%</th>
<th>4.17%</th>
<th>5.26%</th>
<th>6.38%</th>
<th>7.53%</th>
<th>8.70%</th>
<th>9.89%</th>
<th>11.11%</th>
<th>12.36%</th>
<th>13.64%</th>
<th>14.94%</th>
<th>16.28%</th>
<th>17.65%</th>
<th>19.05%</th>
<th>20.48%</th>
<th>21.95%</th>
<th>23.46%</th>
<th>25.00%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup</td>
<td>-2.82%</td>
<td>-9.09%</td>
<td>-13.79%</td>
<td>-16.67%</td>
<td>-20.00%</td>
<td>-23.08%</td>
<td>-28.08%</td>
<td>-30.07%</td>
<td>-32.43%</td>
<td>-34.64%</td>
<td>-36.71%</td>
<td>-38.65%</td>
<td>-40.48%</td>
<td>-42.20%</td>
<td>-43.82%</td>
<td>-45.36%</td>
<td>-46.81%</td>
<td>-47.37%</td>
<td>-47.64%</td>
<td></td>
</tr>
</tbody>
</table>
### Amdahl's Law

Amdahl's Law states that the ultimate speedup of a parallel computer is limited by the fraction of the total execution time that is not parallelizable. Specifically, if a task consists of a fraction \(p\) of serial execution and \(1-p\) of parallel execution, then the maximum possible speedup \(S\) is given by:

\[
S = \frac{1}{p}
\]

In other words, the performance improvement due to parallel processing is directly proportional to the proportion of the task that can be executed in parallel. The higher the proportion of parallelizable code, the greater the potential speedup.

### Example Table

Below is an example table illustrating the impact of Amdahl's Law on the speedup of kernel execution, assuming a baseline execution time of 1.00x.

<table>
<thead>
<tr>
<th>Fraction of Serial Code (p)</th>
<th>Speedup Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00%</td>
<td>50.00x</td>
</tr>
<tr>
<td>0.05%</td>
<td>48.00x</td>
</tr>
<tr>
<td>0.10%</td>
<td>46.00x</td>
</tr>
<tr>
<td>0.15%</td>
<td>44.00x</td>
</tr>
<tr>
<td>0.20%</td>
<td>42.00x</td>
</tr>
<tr>
<td>0.25%</td>
<td>40.00x</td>
</tr>
<tr>
<td>0.30%</td>
<td>38.00x</td>
</tr>
<tr>
<td>0.35%</td>
<td>36.00x</td>
</tr>
<tr>
<td>0.40%</td>
<td>34.00x</td>
</tr>
<tr>
<td>0.45%</td>
<td>32.00x</td>
</tr>
<tr>
<td>0.50%</td>
<td>30.00x</td>
</tr>
</tbody>
</table>

The table shows that as the fraction of serial code increases, the speedup factor decreases, highlighting the diminishing returns of parallel processing.
## Amdahl's Law

Amdahl's Law is a formula used to describe the theoretical speedup over a computer system caused by a change in system design. It is often used in computer science to model the performance of hardware and software systems.

Amdahl's Law can be expressed as:

$$ S = \frac{1}{(1-r) + \frac{r}{n}} $$

Where:
- $S$ is the speedup achieved.
- $r$ is the fraction of the system that is improved.
- $n$ is the number of processors.

### Example

**Make Kernel 2x Faster**

<table>
<thead>
<tr>
<th>$r$</th>
<th>$n$</th>
<th>$S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>1</td>
<td>2.00</td>
</tr>
<tr>
<td>0.5</td>
<td>2</td>
<td>2.00</td>
</tr>
<tr>
<td>0.5</td>
<td>3</td>
<td>2.00</td>
</tr>
<tr>
<td>0.5</td>
<td>4</td>
<td>2.00</td>
</tr>
</tbody>
</table>

This example shows that if 50% of the system is improved, then doubling the number of processors will double the speedup.

### Table

<table>
<thead>
<tr>
<th>$r$</th>
<th>$n$</th>
<th>$S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25</td>
<td>1</td>
<td>2.00</td>
</tr>
<tr>
<td>0.25</td>
<td>2</td>
<td>2.00</td>
</tr>
<tr>
<td>0.25</td>
<td>3</td>
<td>2.00</td>
</tr>
<tr>
<td>0.25</td>
<td>4</td>
<td>2.00</td>
</tr>
</tbody>
</table>

This table demonstrates the effect of improving 25% of the system on the speedup with varying numbers of processors.

### Conclusion

Amdahl's Law is a powerful tool for understanding the limits of hardware and software improvements. It is particularly useful in the design of high-performance computing systems.
### Amdahl's Law

**make kernel 4x faster**

<table>
<thead>
<tr>
<th>Percentage</th>
<th>3.00%</th>
<th>8.11%</th>
<th>12.68%</th>
<th>17.65%</th>
<th>23.08%</th>
<th>29.03%</th>
<th>35.59%</th>
<th>42.86%</th>
<th>50.94%</th>
<th>60.00%</th>
<th>70.21%</th>
<th>81.82%</th>
<th>95.12%</th>
<th>110.53%</th>
<th>128.57%</th>
<th>150.00%</th>
<th>175.86%</th>
<th>207.69%</th>
<th>247.83%</th>
<th>300.00%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup</td>
<td>-1.23%</td>
<td>-2.56%</td>
<td>-1.00%</td>
<td>-1.00%</td>
<td>-1.00%</td>
<td>-2.56%</td>
<td>-1.00%</td>
<td>-2.56%</td>
<td>-1.00%</td>
<td>-1.00%</td>
<td>-2.56%</td>
<td>-1.00%</td>
<td>-1.00%</td>
<td>-1.00%</td>
<td>-1.00%</td>
<td>-2.56%</td>
<td>-1.00%</td>
<td>-1.00%</td>
<td>-1.00%</td>
<td>-1.00%</td>
</tr>
</tbody>
</table>

The table above shows the speedup in performance for different percentage contributions of a kernel. For example, a 3.00% contribution results in a 1.23% speedup, while a 300.00% contribution results in a 1.27% speedup.
### Amdahl's Law

**make kernel 8x faster**

<table>
<thead>
<tr>
<th>Fraction of the Program</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01%</td>
<td>0.9999</td>
</tr>
<tr>
<td>0.1%</td>
<td>0.999</td>
</tr>
<tr>
<td>1%</td>
<td>0.9</td>
</tr>
<tr>
<td>5%</td>
<td>0.95</td>
</tr>
<tr>
<td>10%</td>
<td>0.9</td>
</tr>
<tr>
<td>50%</td>
<td>0.8</td>
</tr>
<tr>
<td>90%</td>
<td>0.5</td>
</tr>
<tr>
<td>99%</td>
<td>0.1</td>
</tr>
<tr>
<td>99.9%</td>
<td>0.01</td>
</tr>
<tr>
<td>99.99%</td>
<td>0.001</td>
</tr>
</tbody>
</table>

As the fraction of the program that can be sped up increases, the overall performance improvement decreases according to Amdahl's Law.
# Amdahl’s Law

The table below demonstrates the potential performance improvements of a kernel by making different components faster. The data is presented as a percentage increase in performance.

<table>
<thead>
<tr>
<th>Percentage Increase</th>
<th>Performance Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.92%</td>
<td>-0.31%</td>
</tr>
<tr>
<td>10.34%</td>
<td>-0.62%</td>
</tr>
<tr>
<td>16.36%</td>
<td>-0.93%</td>
</tr>
<tr>
<td>23.08%</td>
<td>-1.23%</td>
</tr>
<tr>
<td>30.61%</td>
<td>-1.54%</td>
</tr>
<tr>
<td>39.13%</td>
<td>-1.84%</td>
</tr>
<tr>
<td>48.84%</td>
<td>-2.14%</td>
</tr>
<tr>
<td>60.00%</td>
<td>-2.44%</td>
</tr>
<tr>
<td>72.97%</td>
<td>-2.74%</td>
</tr>
<tr>
<td>88.24%</td>
<td>-3.03%</td>
</tr>
<tr>
<td>106.45%</td>
<td>-3.32%</td>
</tr>
<tr>
<td>128.57%</td>
<td>-3.61%</td>
</tr>
<tr>
<td>156.00%</td>
<td>-3.90%</td>
</tr>
<tr>
<td>190.91%</td>
<td>-4.19%</td>
</tr>
<tr>
<td>236.84%</td>
<td>-4.48%</td>
</tr>
<tr>
<td>300.00%</td>
<td>-4.76%</td>
</tr>
<tr>
<td>392.31%</td>
<td>-5.01%</td>
</tr>
<tr>
<td>540.00%</td>
<td>-5.26%</td>
</tr>
<tr>
<td>814.29%</td>
<td>-5.50%</td>
</tr>
<tr>
<td>1500.00%</td>
<td>-5.74%</td>
</tr>
</tbody>
</table>

Make kernel 16x faster.
### Amdahl's Law

<table>
<thead>
<tr>
<th>Fraction</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.05%</td>
<td>59.98x</td>
</tr>
<tr>
<td>0.10%</td>
<td>49.92x</td>
</tr>
<tr>
<td>0.20%</td>
<td>29.50x</td>
</tr>
<tr>
<td>0.50%</td>
<td>9.26x</td>
</tr>
<tr>
<td>1.00%</td>
<td>5.76x</td>
</tr>
<tr>
<td>2.00%</td>
<td>2.26x</td>
</tr>
</tbody>
</table>

- **make kernel 32x faster**
4. The Costs of Externalization
Rough Measurements

Object Churn

- 3000 temps to ingest one document
- 70 temps to turn a SOAP date (as bytes) into a Java Calendar
- 6 temps to turn month into int

Fractal Webs of Invocations

- 200k calls to ingest that document
- 2000 calls to turn an XML timecard into a Java data structure

(Sevitsky 2006)
Three Kinds of Expenses

Optimizable: 43%
- Serialization
- String Operations
- Intra-object Copying

Data Motion: 32%
- Reflection
- Data-driven Dispatching
- Connection Management

Vital: 21%
- Application Logic
- Caching
- Encryption

Amortizable
Does the Story Vary?

All Data Sets:
- 50% Optimizable Data Motion
- 31% Amortizable
- 19% Vital

Analytics Apps:

Web Apps:

SPECjbb2013:
- 33%
- 53% Vital

SPECjEnterprise:
- 49% Optimizable
- 34% Amortizable
- 13%

trade:
- 54% Amortizable
What the JIT Doesn’t Catch
(numbers relative to original w/o JIT optimizations)

- Copies: 0%
- Comparisons: 25%
- ALU: 50%
- Loads/Stores: 75%

original with JIT optimizations

handtuned w/o JIT optimizations

(Xu 2009)
Allocate-Use Separation

How much do we have to inline to have a chance of removing temps and copies?
Method Inlining

<table>
<thead>
<tr>
<th></th>
<th>base J9 inliner</th>
<th>JOLT inliner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse</td>
<td>0.4%</td>
<td>1.9%</td>
</tr>
<tr>
<td>JPetStore on Spring</td>
<td>0.7%</td>
<td>2.5%</td>
</tr>
<tr>
<td>TPCW on JBoss</td>
<td>0%</td>
<td>4.3%</td>
</tr>
<tr>
<td>DaCapo</td>
<td>3.4%</td>
<td>13.3%</td>
</tr>
</tbody>
</table>

Inlining is hard! Small benchmarks don’t reflect difficulties

(Shankar 2008)
Spending Memory
COBOL vs Java

class Student {
    String name;
    BigDecimal score;
}
Student[] data = new Student[40];

01 Student occurs 40
  02 Name       PIC X(10)
  02 Score      PIC 999V99

600 bytes
4492 bytes

(Suganuma 2008)
5. Alternative Marshalling Schemes
Options

- operational form
  - wire form
    - optimize the translation

- operational form
  - wire form
  - transmit less

- operational form
  - wire form
  - transmit less
Options

- Google protobuf, Apache Thrift, Apache Avro, Scala Pickling

- Optimize the translation

- Wire form

- Transmit less
protobuf, avro, thrift

- declaratively specify schema of data
- automatically generate marshallers

```
struct Work {
  1: i32 num1 = 0,
  2: i32 num2,
  3: Operation op,
  4: optional string comment,
}
```
Does it Matter?

relative time to serialize and deserialize an object

(http://ganges.usc.edu/pgroupW/images/a/a9/Serializarion_Framework.pdf)
Does it Matter?

![Graph showing time vs. number of elements for different serialization methods.]

- **Java**
- **Kryo v1**
- **Kryo v2**
- **Scala Pickling**
- **Pickler Combinators**
- **Unsafe Pickler Combinators**

**Free Memory [Bytes]**

**Size [Bytes]**

(Miller 2013)
Experiment: specjbb2013

- Fraction of CPU Spent in Serialization:
  - 0%
  - 10%
  - 20%
  - 30%
  - 40%
  - 50%

Stack Sample Frequency (seconds):
- 10
- 15
- 20
- 25
- 30
- 60
- 90
- 120
- 240
- 300
- 600

- original implementation
- change ~100 lines of code in the marshaller
6. Alternative Storage Schemes
Value Types and Structs

struct Student {
    char[] name;
    int score;
}

pay the header cost once per **record**
Ref Poisoning

struct Student {
    String name;
    int score;
}

…. unless we use a reference type

in which case, we’re back to where we started
Arrays of Records

```java
struct Student {
    char[] name;
    int score;
}
Student[] data = new Student[10];
```

pay the header cost once per array
struct Student {
    string name;
    decimal score;
}
Student[] data = new Student[10];

pay the header cost once per array

data[3].name.charAt(5)

RPC

operational form and wire form
c.f. Cobol

both are fine... until we need to store non-scalar data, i.e. variable-length data
class Student {
    String name;
    BigDecimal score;
}

PROS
- maintains easily serializability
- allows for mix-and-match use of attributes

CONS
- added overhead (start and end pointers)
- unspeakably horrible code
7. Alternative Compilation Schemes and Optimizable Language Kernels
High-level Languages for Targeting FPGAs

- LegUp (U. Toronto)
- Kiwi (Microsoft Research)
- Bluespec (MIT)
- Lime (IBM Research)

make it easier to write computational kernels
asm.js

Javascript subset that is easier to optimize

C/C++

LLVM

Browser JIT with asm.js support
RPython (c.f. Truffle)

Python

subset that is easier to optimize

Human codes interpreter for language X

JIT for language X
class Student {
    String name;
    BigDecimal score;
}

code is easy to maintain, but performance sucks

code is hard to maintain, or impossible to express in the language, but performance is great

code is easy to maintain, but performance sucks
Can we lower from one to the other?
class Student {
    String name;
    BigDecimal score;
}

What is lowering other than … a partial evaluation of serialization?
Partial Marshal

class Student {
    String name;
    BigDecimal score;
}

class StudentTable {
    CharTable names;
    int[] nameStart;
    int[] nameEnd;
    IntTable scores;
    int[] scoreStart;
    int[] scoreEnd;
}

student.getName();
students.getName(i);

names.splice(students.nameStart[i],
             students.nameEnd[i]);

transformer??

transformer??

students.getName(i);

names.splice(students.nameStart[i],
             students.nameEnd[i]);
Partial Marshal

class Student {
    String name;
    BigDecimal score;
}

class StudentTable
    CharTable names;
    int[] nameStart;
    int[] nameEnd;
    IntTable scores;
    int[] scoreStart;
    int[] scoreEnd;
}

data model

student.getName();

students.getName(i);

names.splice(students.nameStart[i],
             students.nameEnd([i]));
Attic