Research Division

Understanding Performance in Large-scale Framework-based Systems

Gary Sevitsky, Nick Mitchell, Harini Srinivasan
Intelligent Analysis Tools Group
IBM TJ Watson Research Center

April 18, 2005

Background

- Our group develops techniques for understanding Java application behavior
  - performance and memory diagnosis tools
    - e.g. Jinsight (mature tool), LeakBot, JaVinci (ongoing research)
  - characterizing complexity
- Focus is on large framework-based systems
  - high-volume web-based (e-Business) applications
  - client-side applications (e.g. Eclipse-based)
- We maintain a consulting practice, solving problems for IBM customers and products

Observations

- Things are getting worse
  - Performance errors are easy to make
  - Performance errors are difficult to localize
    - and to understand, communicate, assess
  - Costs of design choices are difficult to predict
  - Tools are at the wrong level
  - Automated optimizations are not keeping up
- Even well-tuned programs seem bloated
  - They seem to be doing a lot of work to accomplish very little

Goals for this talk

- Scare you … with how bad things are

Goals for this talk

- Share our experiences from the real world
  - What types of problems occur in these applications?
  - Explore some of the reasons they occur
  - Show some requirements for analysis
    - Illustrate LiveJinsight approach, including its limitations
  - Gain insights on where optimizations are failing

Road Map

- Background: large-scale object-oriented systems
- Case studies in Java performance analysis
  - Part I: Performance errors from the real world
  - Part II: A “well-tuned” benchmark
- Ongoing research
  - Automating performance understanding
  - Characterizing complexity

Object-oriented Design and Java

- Modern O-O design techniques and the Java language aim to ease programming and maintenance, improve correctness, and enable reusability
  - e.g. implementations are hidden behind well-defined interfaces
  - e.g. design patterns distribute functional responsibility
  - e.g. Java provides high-level features like automatic garbage collection, multithreading, and object serialization
- In general, the programmer is free not to worry about what's happening behind the scenes
- These techniques have been very successful, and they have enabled the construction of larger, more complex systems. However ...

Object-oriented Design, Java and Performance

- ... many of these same properties can make performance difficult to predict, performance errors easy to make, and runtime behavior complex to analyze
- Some properties of well-designed Java programs:
  - implementation choices are hidden
  - implementation is functionally distributed across many classes for a single user-level feature
  - many interacting parts; many small methods
  - APIs that return new objects
  - reusable libraries/frameworks from vendors or other teams

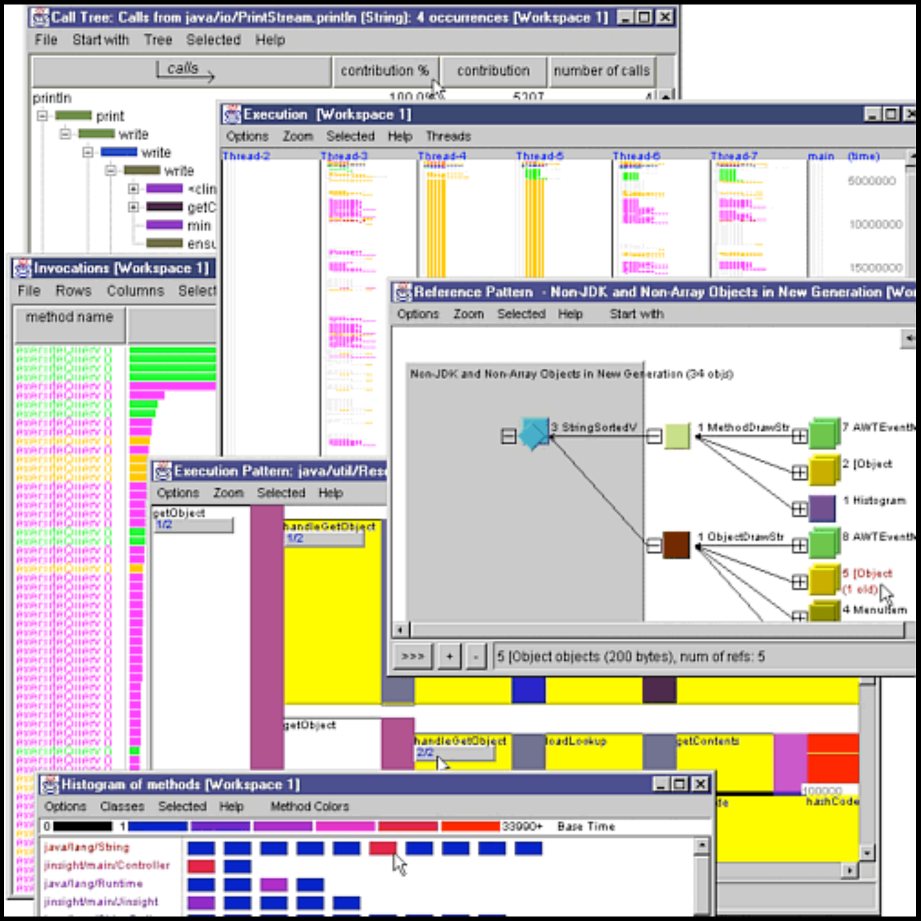
e-Business Applications: Java part

- Extensive use of libraries and frameworks
  - Application server frameworks provide commonly needed services: security, availability, session management, resource pooling, etc.
  - Incredible number of different standards: servlets, JSPs, JDBC, EJBs, JNDI, XML, XSLT, RMI/IIOP, etc.
  - Many are different in kind
  - Most were designed separately, for general-purpose usage
  - Each has its own type system, conventions, etc.
  - Customers have their own frameworks which are reused across applications
  - Many authors, many vendors
- Application itself is usually relatively small
  - and the actual business and presentation logic is relatively simple!
- On the client side we are seeing a similar story (Eclipse, Hyades, eMF, etc.)

Part I: case studies

Jinsight: Understanding Java Application Behavior

- Visual approach
  - For performance and memory analysis
- Traces *details* of an execution
  - Shows *how* and *when* problems occur
  - Allows ad hoc computation of highly focused measures
- Scalable to very large applications
  - Selective and conditional tracing
- Flexibility in navigating and exploring
- Traces using a JVMPI profiling agent
  - Analyze during or after the run
  - Windows, AIX, z/OS, z/Linux, etc.



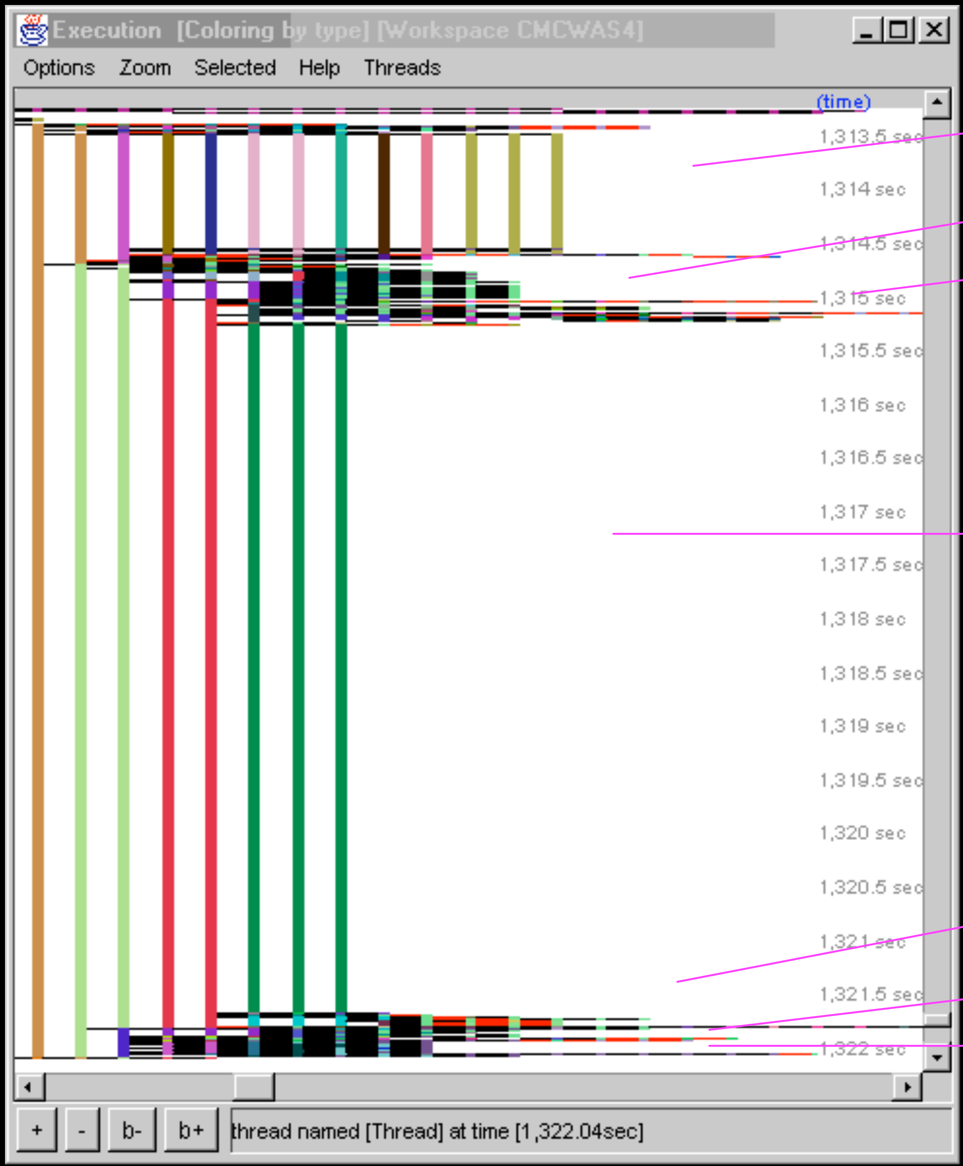
A word of warning about the case studies

- It's easy to think of each case as just another example of bad programming
  - But many of the errors were made by very good programmers!
  - Instead, we would like to encourage questioning of why these problems are so prevalent

Case study #1: Banking application

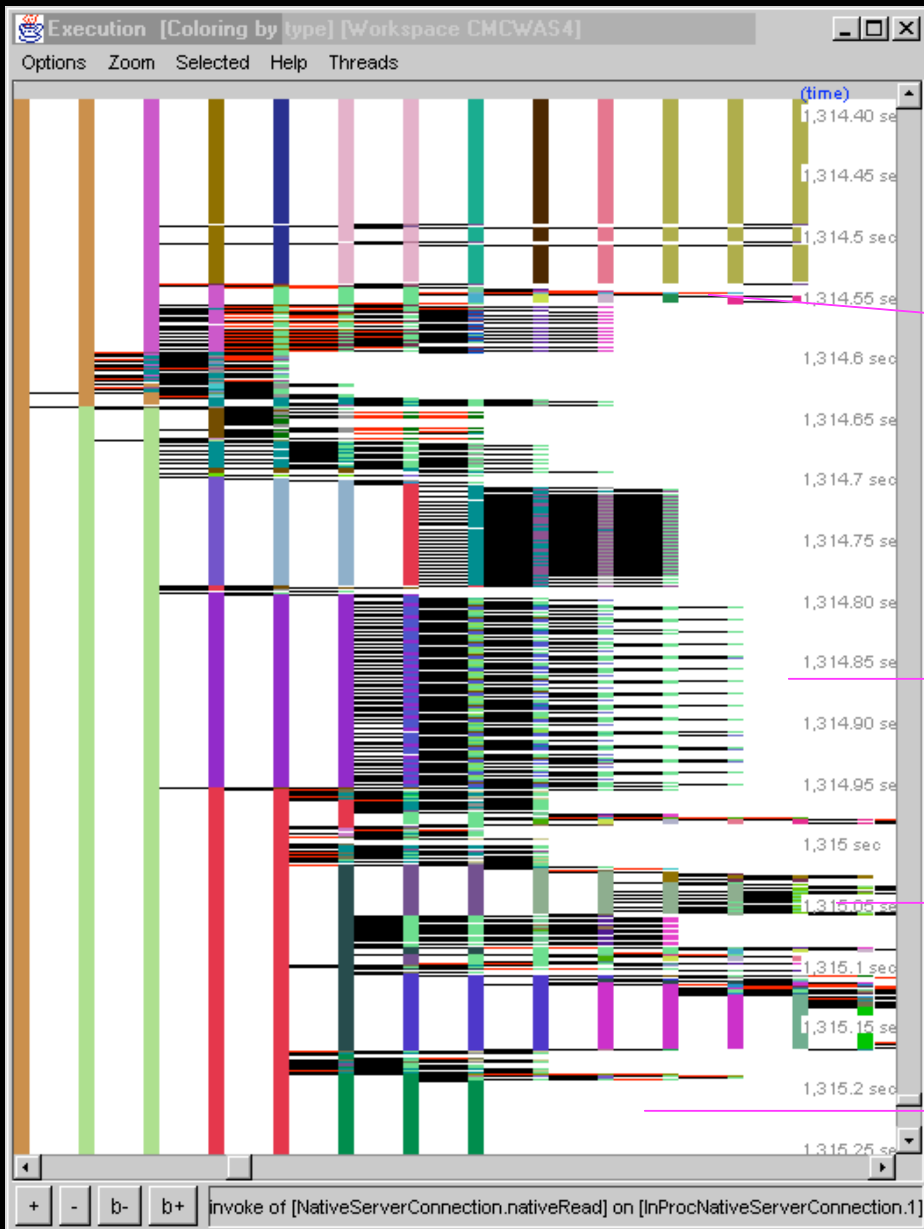
- Large European bank
- Client-server architecture
  - Server: z/OS (IBM 390), WebSphere, additional higher-level frameworks
  - Java client
- Problem: CPU utilization was too high on the server
- Cause: 6-7 independent problems

Anatomy of a transaction



- Get client request
- Parse client request
- Build IMS request
- Send to IMS
- Parse IMS response
- Build client response
- Send client response

Transaction detail: part I



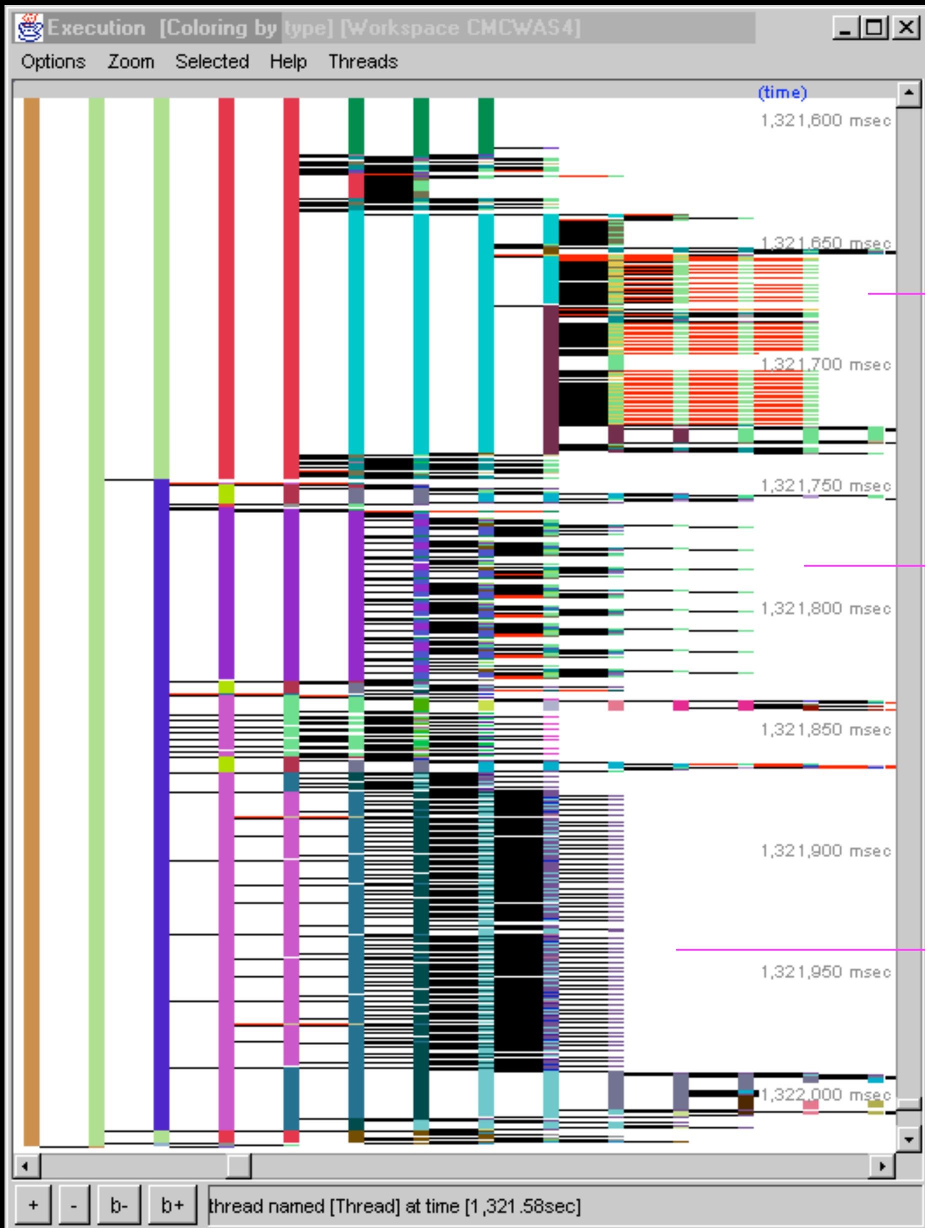
Get client request

Parse client request

Build IMS request

Send to IMS

Transaction detail: part II

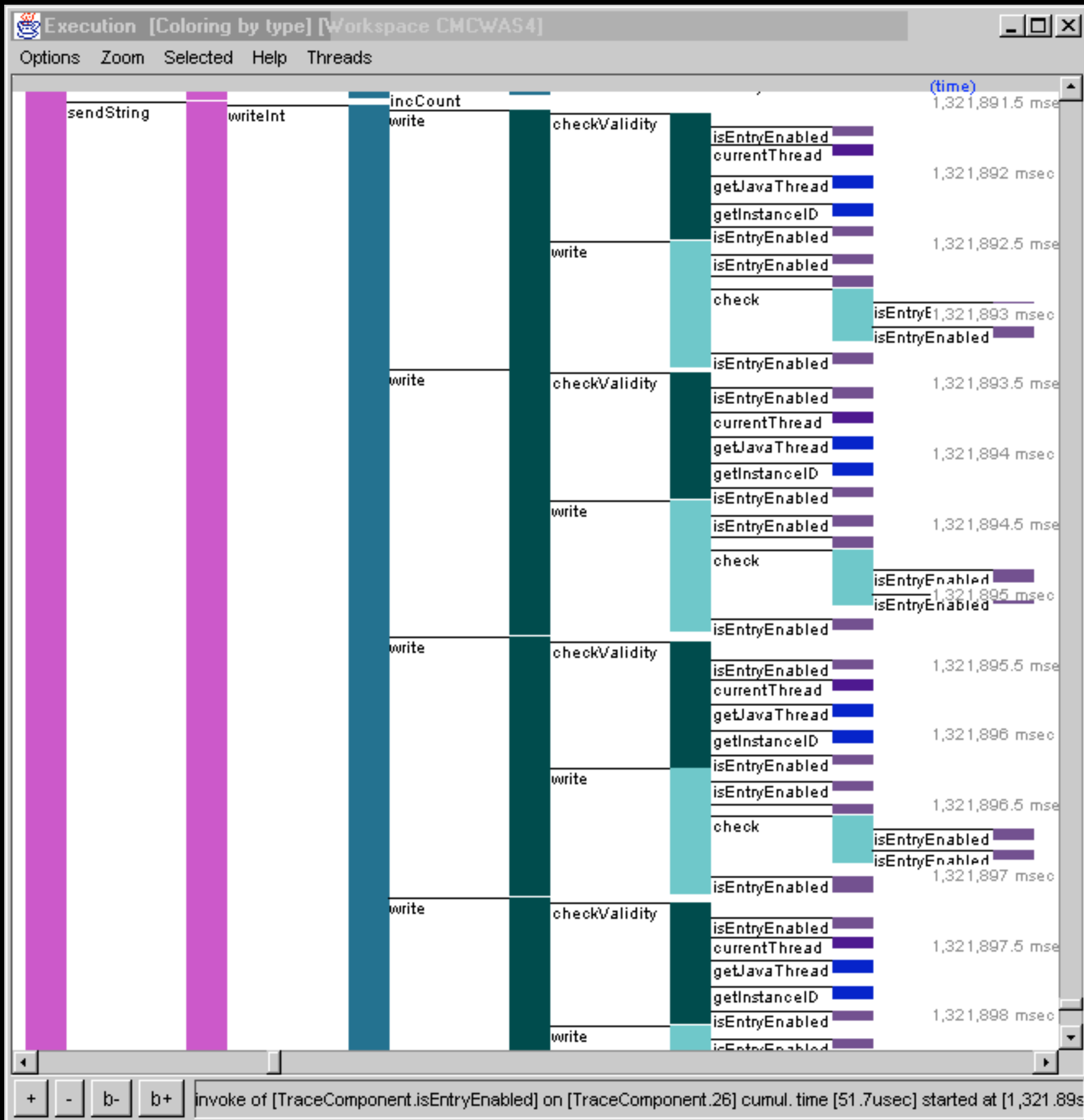


Parse IMS response

Build client response

Send response

Send response (partial)



Send response: detail

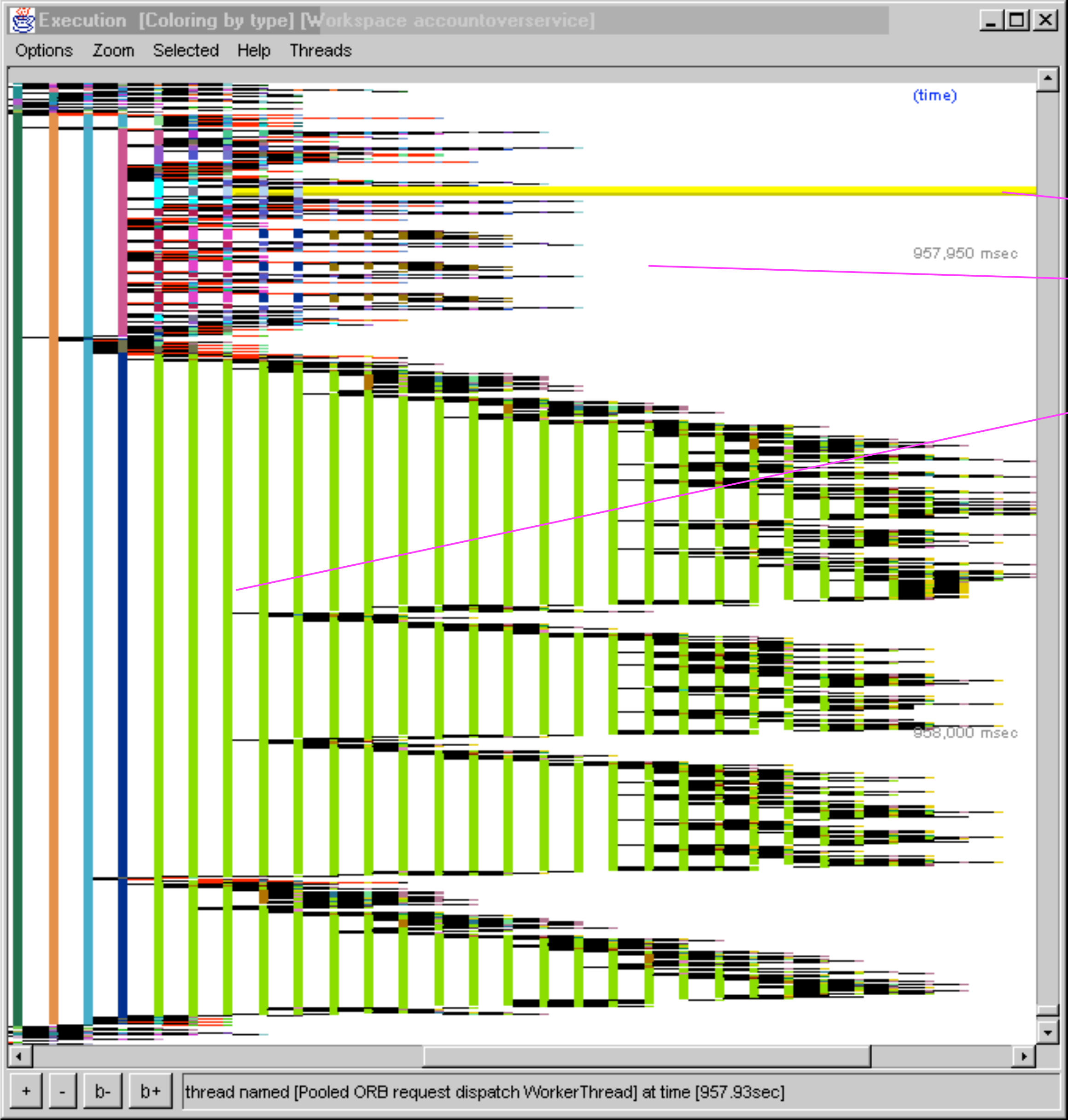
- View shows writing just the header of one String!
- Protocol implemented by layering DataOutputStream over SRTOutputStream
- Solution: buffering

Sending response: summary

- Problem: sending response is expensive
- Solution: introduce a buffering layer
- Comments
  - Performance is not automatically composable!
  - The problem occurred within a framework, and was discovered during application deployment
    - Lesson: performance testing with real-world use cases is especially important for frameworks
  - There actually was a buffering layer, but at the wrong level

Case study #2: Brokerage application

- Web-based client
  - Customer can look up account information, stock holdings
- Server:
  - WebSphere
  - Application uses JSPs, EJBs
  - Customer wrote a general framework to support many applications
- Problem: slow response time
- Cause: at least 3 different problems



Problem: database requests?

actual query to database

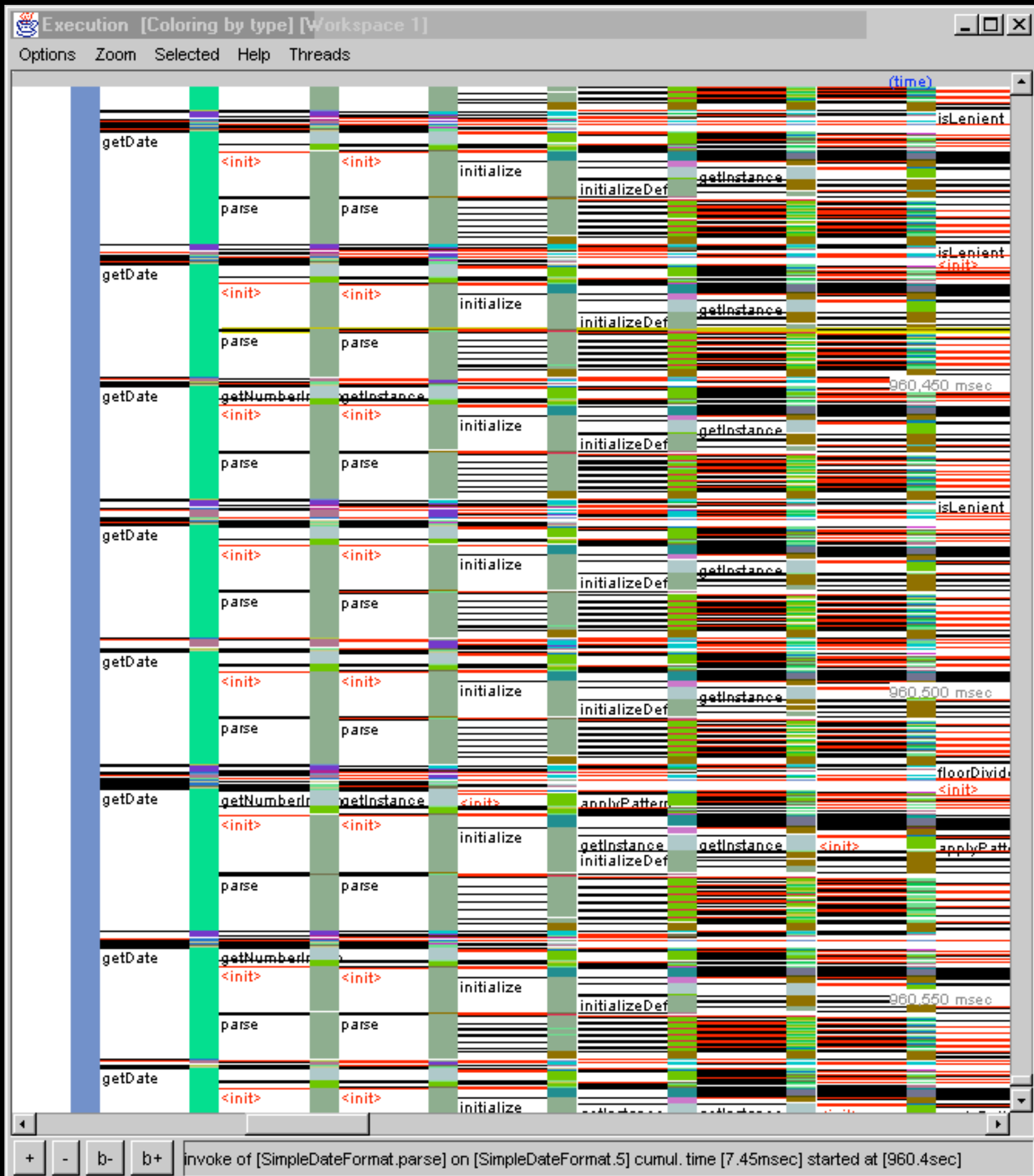
processing 3 records

all the rest: customer LittleInstrumenter class

- uses Java serialization to wrap up database results, just to measure & log their size

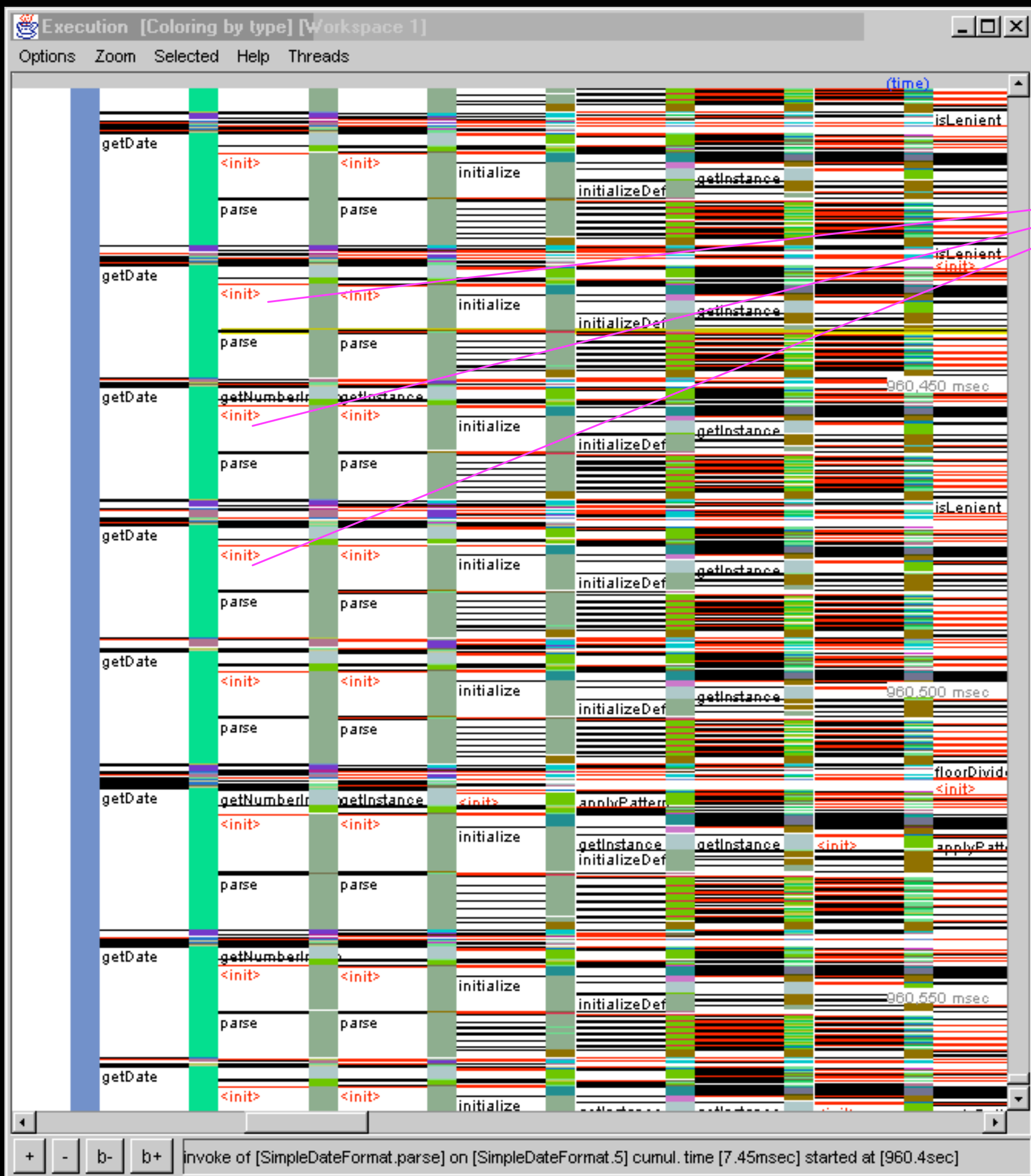
Database request example: summary

- Problem: customer reported that database requests were slow
  - the actual problem was expensive logging, using object serialization
- What went wrong?
  - costs are hidden
    - just one little call to writeObject!
  - *Little*Instrumenter? the code was put in to prevent a performance problem!
  - later, the customer erroneously told us they had fixed the code, yet the problem remained
    - lesson: validation is essential
- Diagnosis techniques
  - Visualization and focused summarization of information in context were key to discovering and measuring the impact of the problem
  - Used data flow (by hand!) to understand the purpose of the serialization



Problem: converting dates

- 63 dates converted to Java format in this one phase of one servlet hit
- cost of converting one date:
  - 1520 method calls
  - 120 temporary objects created



Converting dates

Creating a new SimpleDateFormat each time

- yet the format is always the same!
- also, substantial setup cost to call SimpleDateFormat constructor

Solution: cache the converter

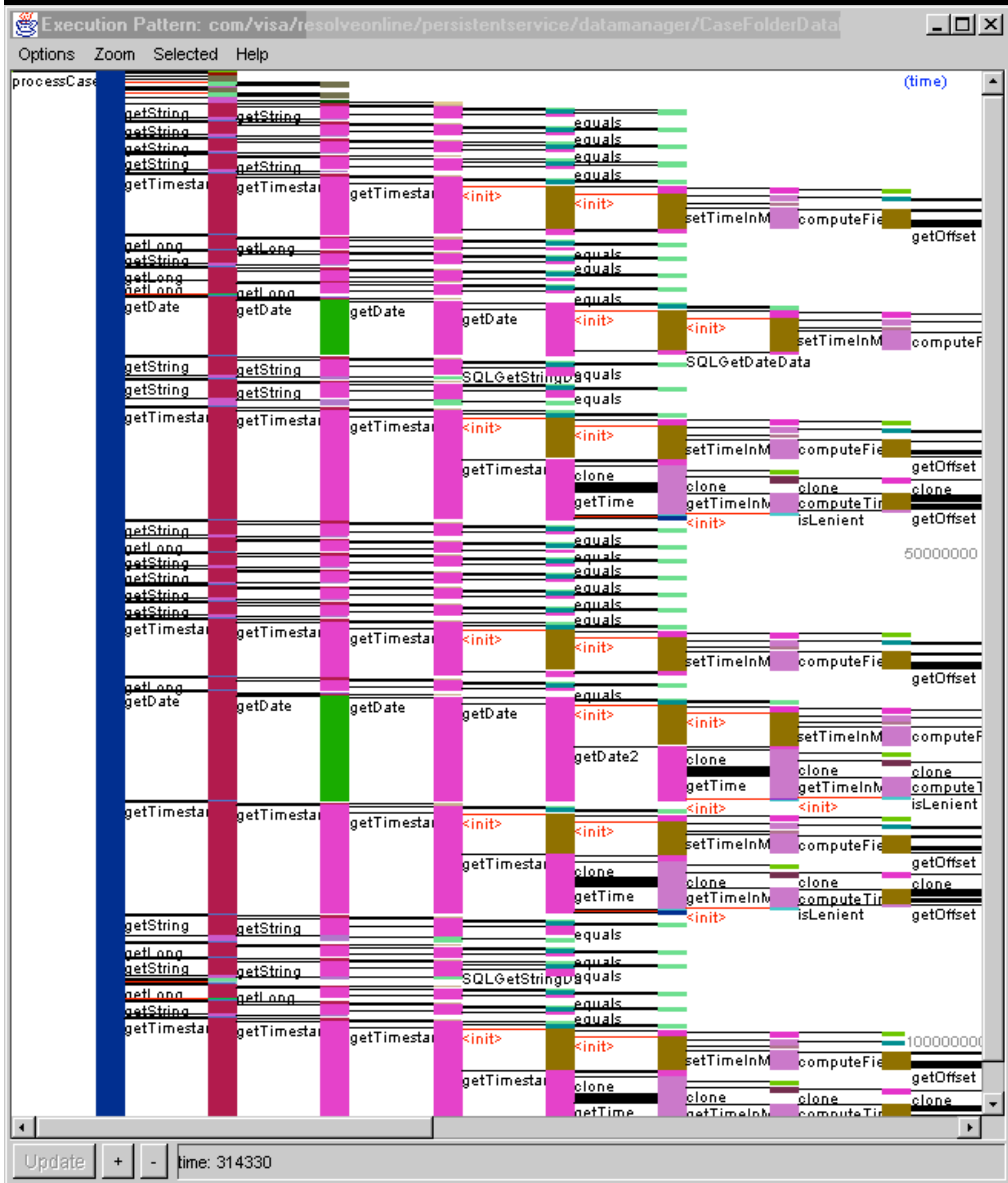
- even once per transaction would help

Date conversion example: summary

- Problem: creating the same object over and over
  - plus an additional setup cost
- Illustrates three common phenomena:
  - recomputation is one of the most common problems
    - the cost of calls is not obvious
  - creation of temporary objects
    - allocation and GC cost are just part of the problem
    - the real expense is initializing temporary data structures
  - note that the remaining part of the conversion is still expensive
    - conversion is a major expense even in “correctly-written” applications
- Diagnosis techniques
  - understanding and focused summarization of activity in a particular context were essential to discovery and accurate measurement of the impact
  - data flow and escape information (guessed at, by hand) were valuable for understanding

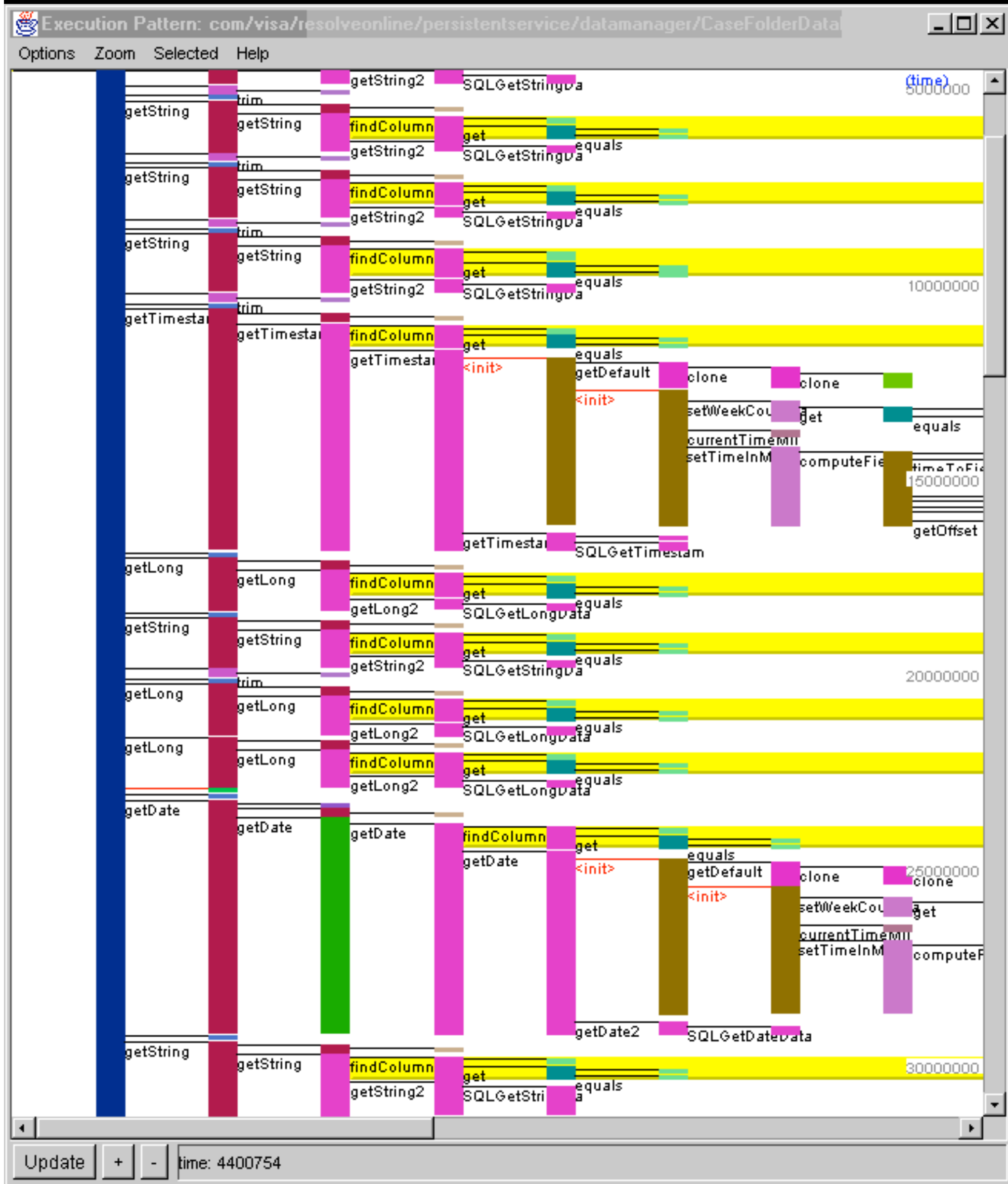
Case study #3: Credit card application

- Problem: slow response time
- Causes: many different problems (only 2 shown here)



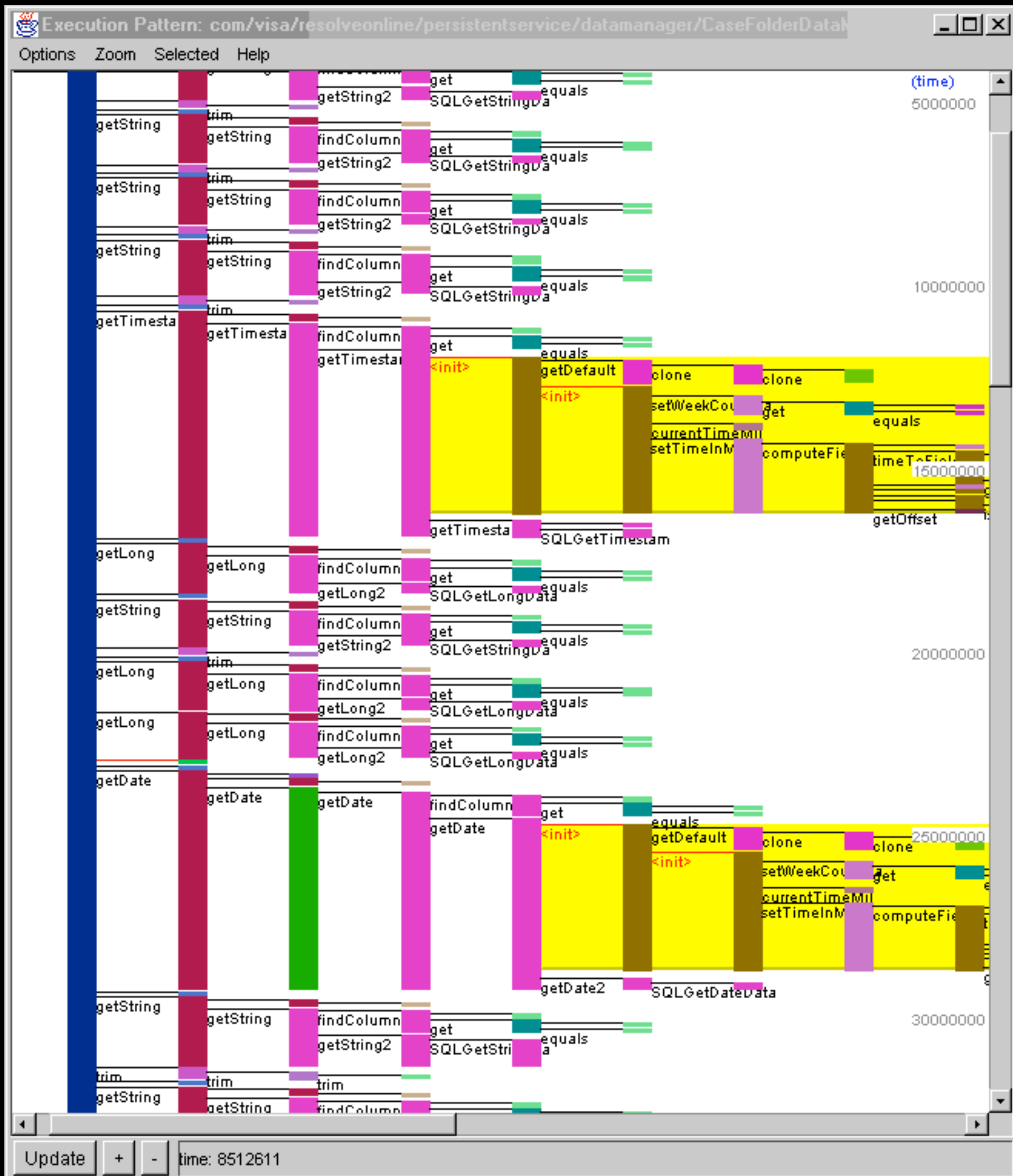
Database requests

- View shows getting the field values out of *one* row
- Called in a loop (e.g. 25 records for one query)
- One row costs:
  - 728 method calls
  - 106 new short-lived objects
  - after JIT!



Database requests

- Calls these ResultSet methods:
  getString (String columnName)
  getLong (String columnName)
  etc.
- Causes findColumn(columnName) to be called for each field
  - but the column structure is fixed for every record!
- Solution: use different calls:
  getString (int columnIndex)
  getLong (int columnIndex)
  etc.



Database requests

- Part 2: getDate and getTimestamp cause new GregorianCalendar to get created each time
- Solution: use different calls:
  getDate (columnIndex, Calendar)
  getTimestamp (columnIndex, Calendar)
- But these calls were not implemented in the DB2 driver!
- What went wrong?
  - Knowledge of correct API required
    - Causing unnecessary recomputation & object re-creation
  - Part 2: driver implementation not suited for common use case

```
private static KeyFactory instance = new KeyFactory();


// Create a unique credit card transaction key
public synchronized long getSIDKey() {
    try {
        Thread.sleep(1);
         long key =
             expression based on current time and server name
         return key;

    } catch(Exception e) {
          return getSIDKey();
    }
}


in a loop within each servlet hit:
instance.getSIDKey();
```

Create transaction key

Sleep in a synchronized method

- Contention problem
- Response time problem

Called 17 times in one hit!

```
private static KeyFactory instance = new KeyFactory();


// Create a unique credit card transaction key
public synchronized long getSIDKey() {
    try {
         Thread.sleep(1);
          long key =
               expression based on current time and server name
          return key;

    } catch(Exception e) {
           return getSIDKey();
    }
}

in one servlet hit, in a loop:
…
instance.getSIDKey();
```

Create transaction key

Want went wrong?

- Just “coding crazy”?
- Rather, it was insufficient awareness of scalable multithreading issues

Recursion in exception handler?

Part II: The Diary of a Datum

Ongoing Research

JaVinci: Automated Performance Explanation

- Problem: current tools place too much burden on the user
  - Too much expertise is required to interpret the data
  - Too much work is required to dig through details, even for experts
- Goal: simplify performance diagnosis and understanding
  - *Challenge: can we turn a 500K method call trace into a manager's summary?*
- Approach:
  - Build collective expertise into the tools
    - Knowledge about how problems are analyzed
    - Domain knowledge (e.g. about J2EE, WCS)
    - Knowledge of what is worth tracing
  - Let the system do the hard work: automate much of analysis and trace collection
  - Raise the level of explanation
  - Integrate many layers of explanation
  - Combine static and dynamic analyses

Characterizing Complexity

- Goal: Understand the nature and causes of run-time complexity
- Enables:
  - Performance understanding and assessment of individual applications
  - Comparisons across various implementations
  - Characterization of classes of applications
    - Identify good API design practice
    - Identify classes of optimizations to target
- FSE 2005 submission

People

- Customer examples; Descriptive characterization
  - Nick Mitchell, Gary Sevitsky, Harini Srinivasan
- Jinsight (past)
  - Wim De Pauw, Herb Derby, Olivier Gruber, Erik Jensen, Ravi Konuru, Martin Robillard, Gary Sevitsky, Harini Srinivasan, John Vlissides, Jeaha Yang
- JaVinci: automation of performance understanding
  - Gary Sevitsky, Nick Mitchell
  - Barbara Ryder