Software Visualization

Wesley Coelho

CPSC 533C

March 29, 2004

Visualizations for Software Engineering

- Visualizations for the following engineering tasks are reviewed:
  - Optimization
  - Testing
  - Monitoring deployed software
- Common themes
  - Overview + detail views
  - Source code is abstracted with SeeSoft views (Eick, Steffen and Sumner, 1992)

Reviewed Papers

- Visualizing Application Behavior on Superscalar Processors (Stolte, Bosch, Hanrahan and Rosenblum, 1999)
- Technical Note: Visually Encoding Program Test Information to Find Faults in Software (Eagan, Harrold, Jones and Stasko, 2001)
- Visualization of Program-Execution Data for Deployed Software (Orso, Jones and Harrold, 2003)

Introduction

- Goal: Visualize program instruction execution on a superscalar processor
- Superscalar processors
  - Can execute more than one instruction per cycle
  - Instructions can be executed out-of-order
  - Some instructions depend on the results of other instructions
- Program source code structure can be modified to increase instruction-level parallelism for better performance

Why Visualize?

- Software developers rarely attempt such optimizations
  - Individual instructions need to be investigated
  - Millions of instructions are executed per second
  - Programmers work with source code, not instructions

Sample Dataset

```
PC:401eb8 IHI:4d ILO:   40418 ;sra       r4,r4,24
PC:401ec0 IHI: 3 ILO:  1007f6 ;jal       0x401fd8
PC:401fd8 IHI:49 ILO:   40418 ;sll       r4,r4,24
PC:401fe0 IHI:4d ILO:   4040e ;sra       r4,r4,14
PC:401fe8 IHI:71 ILO:   110e5 ;lui       r1,0x10e5
PC:401ff0 IHI:36 ILO: 4010100 ;addu      r1,r4,r1
PC:401ff8 IHI:15 ILO: 100c1e8 ;l.d       f0,-15896(r1)
PC:402000 IHI:76 ILO: 2060000 ;dmtc1     r6,f2
PC:402008 IHI:36 ILO:     600 ;addu      r6,r0,r0
PC:402010 IHI:6a ILO:   20000 ;c.lt.d    f0,f2
PC:402018 IHI:37 ILO:   7007f ;addiu     r7,r0,127
PC:402020 IHI: c ILO:       8 ;bc1f      0x402048
PC:402048 IHI:36 ILO:     500 ;addu      r5,r0,r0
PC:402050 IHI:71 ILO:   210e5 ;lui       r2,0x10e5
PC:402058 IHI:37 ILO: 202bdf0 ;addiu     r2,r2,-16912
PC:402060 IHI:36 ILO: 4020400 ;addu      r4,r4,r2
PC:402068 IHI:36 ILO: 6070200 ;addu      r2,r6,r7
PC:402070 IHI:4d ILO:   20301 ;sra       r3,r2,1
```

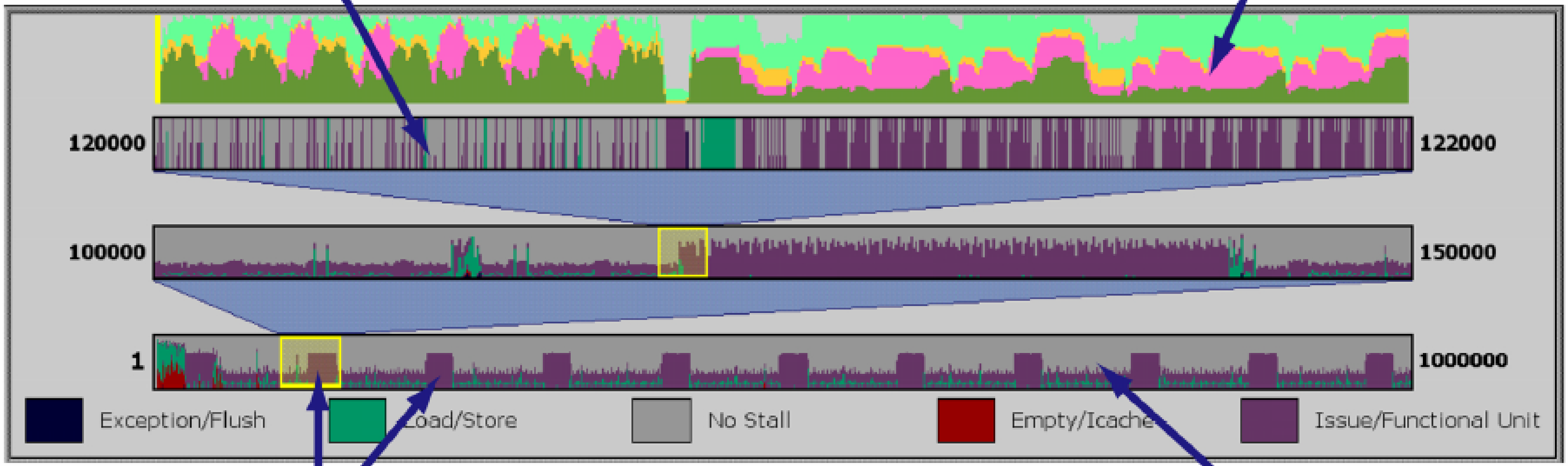
Visualization Approach

- Overview + Detail display based on three views
  - Timeline View
    - Overview of application’s execution
    - Used to find problems
  - Pipeline View
    - Detailed view of instructions in the pipeline at a particular cycle
    - Used to identify a problem
  - Source Code View
    - Relates overview and detail views to lines of source code

Timeline View

③ We are able to focus the area of interest to 2000 cycles -- few enough cycles that we can use animation for further investigation.

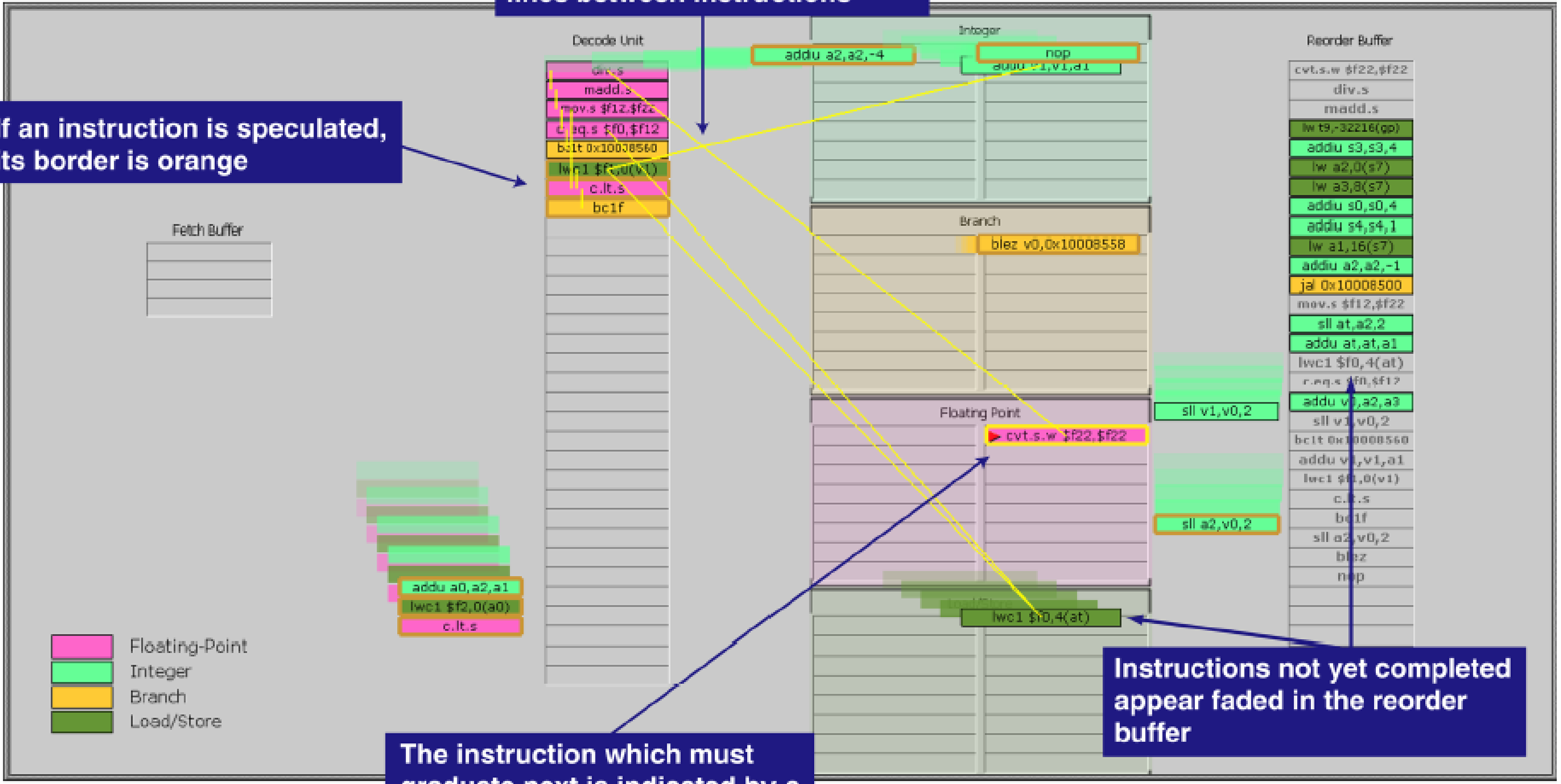
④ The instruction mix chart lets us see what types of instructions are in the pipeline during the time interval of interest.



② There are periods of increased pipeline stall throughout the execution

① The overview displays stall and throughput information for the entire execution.

Pipeline View



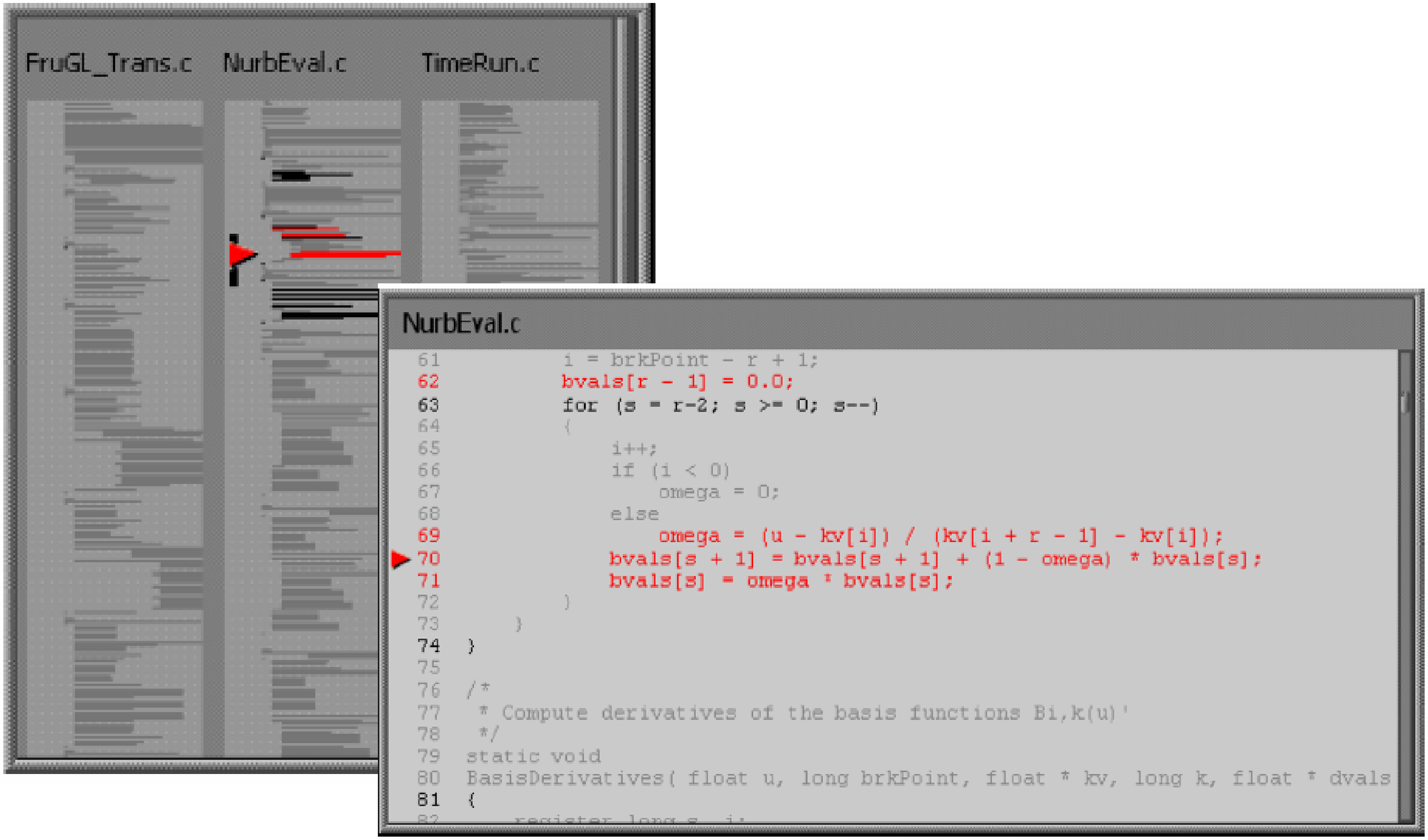
Dependencies appear as yellow lines between instructions

If an instruction is speculated, its border is orange

The instruction which must graduate next is indicated by a yellow border and red arrow

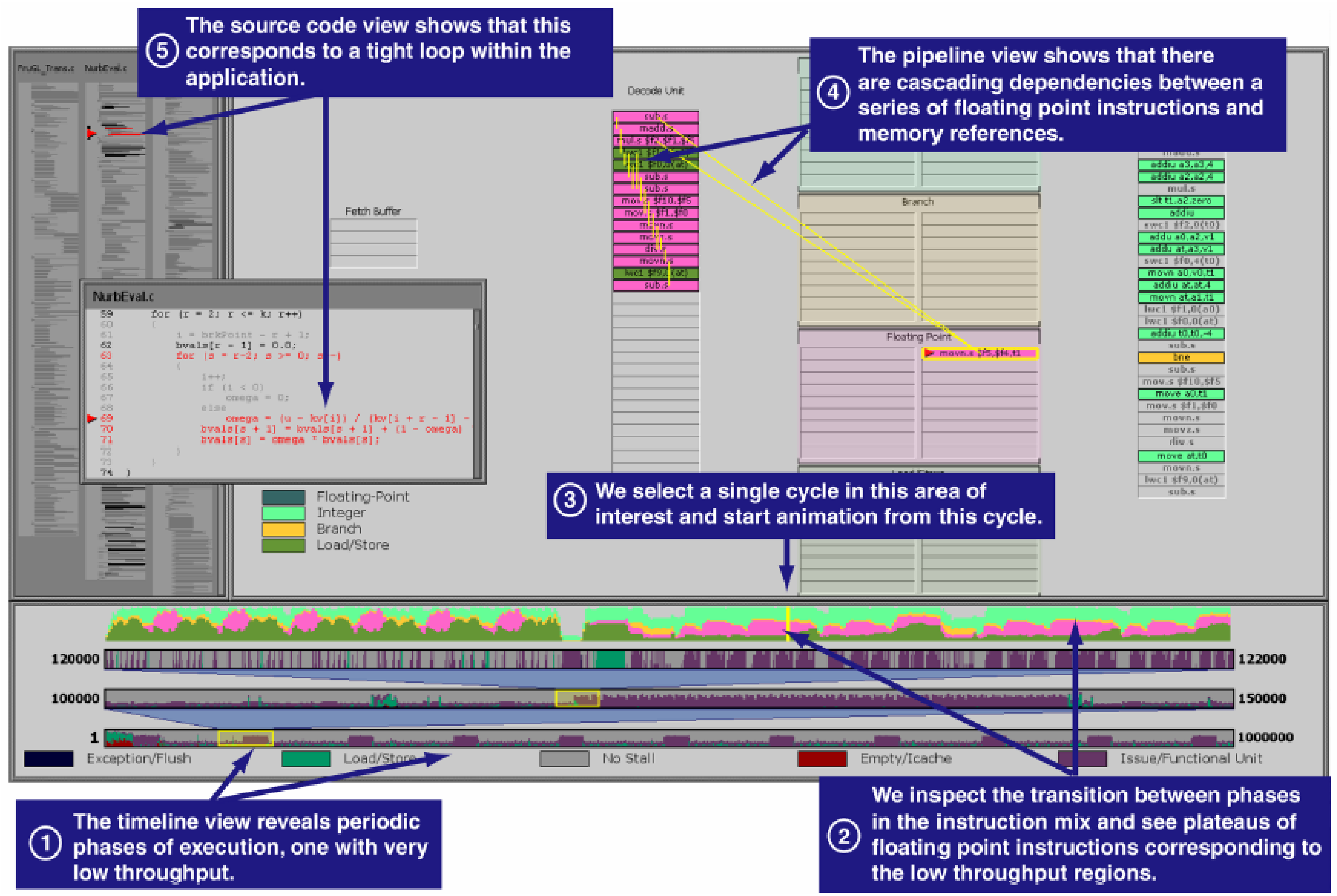
Instructions not yet completed appear faded in the reorder buffer

Source Code View



```
        i = brkPoint - r + 1;
        bvals[r - 1] = 0.0;
        for (s = r-2; s >= 0; s--)
        {
            i++;
            if (i < 0)
                omega = 0;
            else
                omega = (u - kv[i]) / (kv[i + r - 1] - kv[i]);
            bvals[s + 1] = bvals[s + 1] + (1 - omega) * bvals[s];
            bvals[s] = omega * bvals[s];
        }
    }
}

/*
 * Compute derivatives of the basis functions Bi,k(u)'
 */
static void
BasisDerivatives( float u, long brkPoint, float * kv, long k, float * dvals
{
    register long s, i;
```



NurbEval.c

```
    for (r = 2; r <= k; r++)
    {
        i = brkPoint - r + 1;
        bvals[r - 1] = 0.0;
        for (s = r-2; s >= 0; s--)
        {
            i++;
            if (i < 0)
                omega = 0;
            else
                omega = (u - kv[i]) / (kv[i + r - 1] -
            bvals[s + 1] = bvals[s + 1] + (1 - omega) *
            bvals[s] = omega * bvals[s];
        }
    }
}
```

① The timeline view reveals periodic phases of execution, one with very low throughput.

② We inspect the transition between phases in the instruction mix and see plateaus of floating point instructions corresponding to the low throughput regions.

③ We select a single cycle in this area of interest and start animation from this cycle.

④ The pipeline view shows that there are cascading dependencies between a series of floating point instructions and memory references.

⑤ The source code view shows that this corresponds to a tight loop within the application.

Paper Critique

- Strengths
  - These techniques are general enough for use in other applications: Compiler and hardware design, assembly lines, graphics pipelines
  - Animation could be very useful for understanding pipeline behaviour
  - Intuitive use of visual cues in timeline view
  - Self contained – accessible background information about superscalar processors is included
- Weaknesses
  - Scalability -- Only one second of instructions can be visualized
  - Description of animation is deferred to another paper
  - Somewhat complicated colouring scheme for instructions in pipeline view, no legend for instruction border colours
  - Fixed timeline intervals, no explanation for chosen values
  - No explanation of how mapping from instructions to source lines is performed, or what input data is required

Reviewed Papers

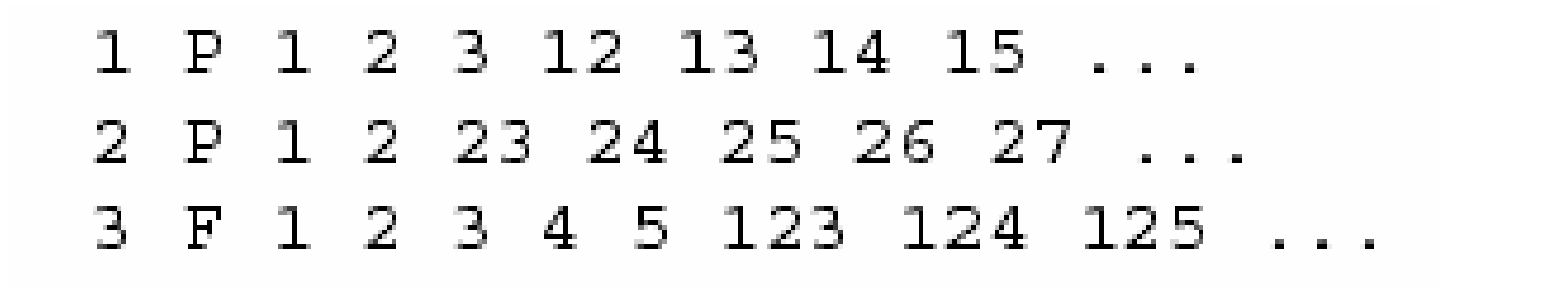
- Visualizing Application Behavior on Superscalar Processors (Stolte, Bosch, Hanrahan and Rosenblum, 1999)
- Technical Note: Visually Encoding Program Test Information to Find Faults in Software (Eagan, Harrold, Jones and Stasko, 2001)
- Visualization of Program-Execution Data for Deployed Software (Orso, Jones and Harrold, 2003)

Tarantula

- A visualization for automated software test suite results
- Large systems sometimes have thousands of test cases
- Tarantula provides a high-level overview of how the software functions under testing

Input Dataset

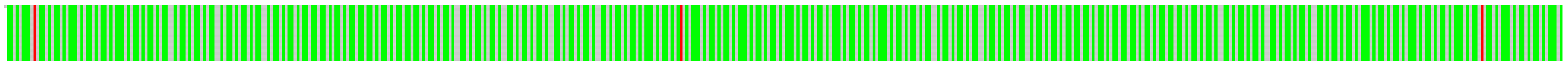
- Test case results
  - Test number
  - Pass or Fail
  - Lines of code covered during test execution

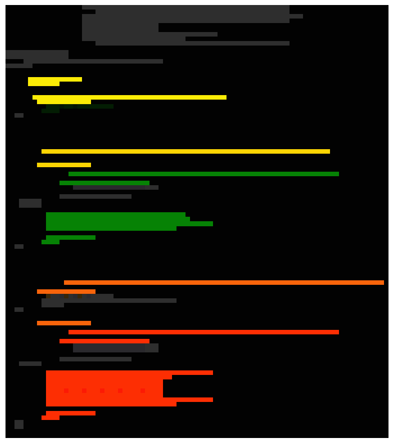


```
1 P 1 2 3 12 13 14 15 ...
2 P 1 2 23 24 25 26 27 ...
3 F 1 2 3 4 5 123 124 125 ...
```

Visualization Approach

- Overview of test results is shown with an array of rectangles representing test cases executed
  - Green rectangles indicate passed tests
  - Red rectangles indicate failed tests

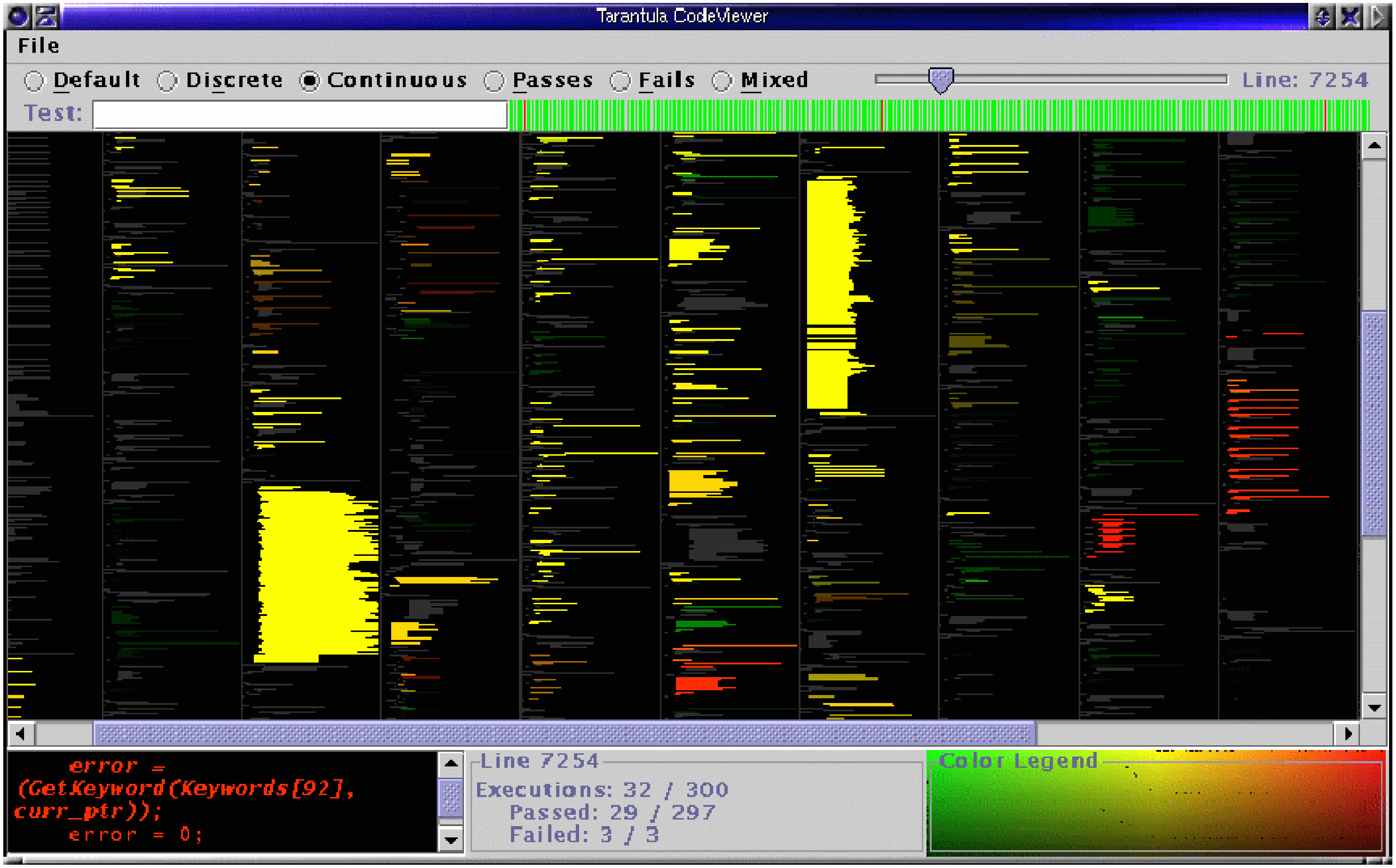




- Lines representing source-code lines are coloured to indicate the number of passed or failed tests that executed that line

Source-line colouring scheme

- Hue is displayed on a spectrum from red to yellow to green
  - More red indicates the statement was executed in a higher proportion of failed tests
- Brightness indicates the number of tests that executed the statement
  - High brightness indicates a high number of tests that executed the statement passed or failed
- Intuition: Lines that are most likely to be faulty should be closer to bright red



Default Discrete [✓ Continuous] Passes Fails Mixed

```
    error =
(GetKeyword(Keywords[92],
curr_ptr));
    error = 0;
```

Line 7254
Executions: 32 / 300
Passed: 29 / 297
Failed: 3 / 3

Paper Critique

- Strengths
  - This is a useful solution to a real problem
  - Paper explains why several simpler colouring schemes were not used
  - Flexible interface, i.e. “Discrete Mode” available for a simpler perspective of the faults
- Weaknesses
  - Source code window is too small. May be difficult to scroll if code changes when you mouse over the main view to get to the scrollbar
  - The name of a file containing a selected source code line is not shown
  - Colour Legend could include axis labels indicating what bright red or dark yellow means
  - Confusing description of the actual meaning of the Hue and Brightness colouring scheme
  - Is there a system available for producing the input to this tool?
  - Scalability – System can only show results for a few files at a time

Reviewed Papers

- Visualizing Application Behavior on Superscalar Processors (Stolte, Bosch, Hanrahan and Rosenblum, 1999)
- Technical Note: Visually Encoding Program Test Information to Find Faults in Software (Eagan, Harrold, Jones and Stasko, 2001)
- Visualization of Program-Execution Data for Deployed Software (Orso, Jones and Harrold, 2003)

Motivation and Dataset

- Many software problems arise only when deployed
- The Gamma tool is capable of collecting program-execution data
  - Coverage data
  - Exception-related information
  - Profiling information
  - Memory and CPU usage
- This can produce a vast amount of data when there are many deployed instances

Gammatella

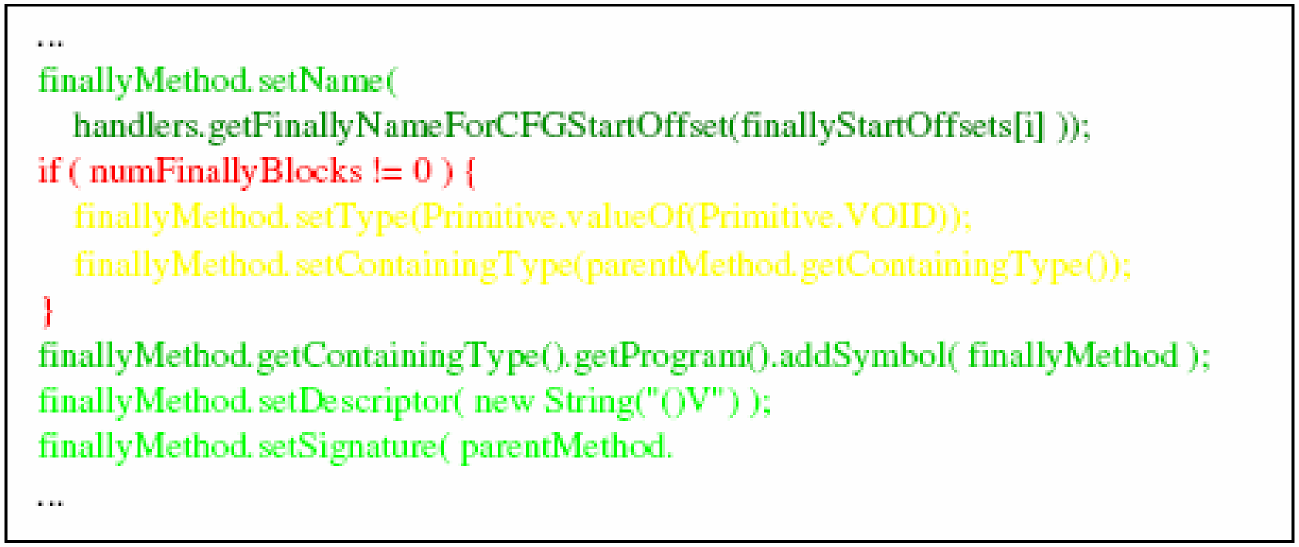
- Implements a novel approach for visualizing program-execution data
- Supports continuous monitoring and exploration
- Program-execution data is shown by applying colour to different levels of program representation
  - Statement Level
  - File Level
  - System Level

Example Application: Profiling

- Profiling finds code that is executed often
- This is useful for
  - Finding code to optimize
  - Determining feature usage
  - Reducing software bloat
- Colour assignment
  - Red = statement executed very often
  - Yellow = statement executed often
  - Green = statement executed rarely

Statement Level

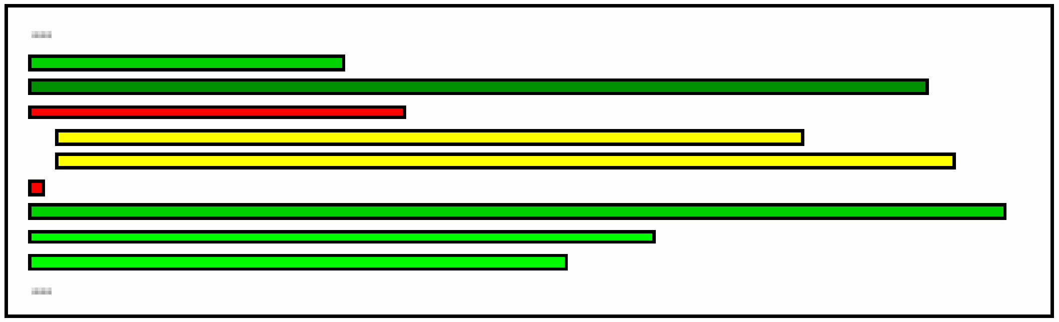
- Provides detail by showing actual source code
- Higher levels of are abstraction required



```
...
finallyMethod.setName(
   handlers.getFinallyNameForCFGStartOffset(finallyStartOffsets[i] ));
if ( numFinallyBlocks != 0 ) {
   finallyMethod.setType(Primitive.valueOf(Primitive.VOID));
   finallyMethod.setContainingType(parentMethod.getContainingType());
}
finallyMethod.getContainingType().getProgram().addSymbol( finallyMethod );
finallyMethod.setDescriptor( new String("()V") );
finallyMethod.setSignature( parentMethod.
...
```

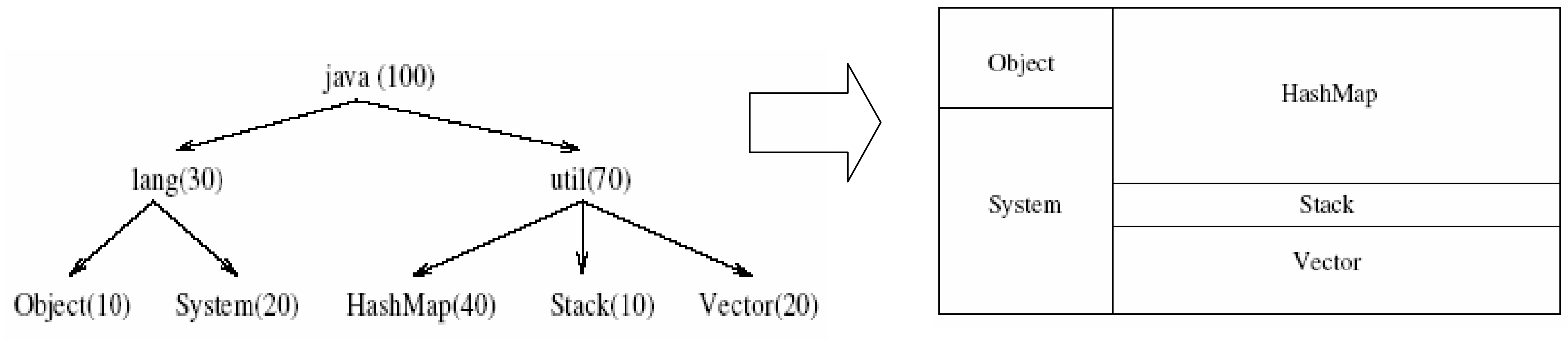
File Level

- SeeSoft-style miniature view of source code
- Relative colours of source code lines still visible
- Still not suitable for viewing large programs



System Level Treemap

- The system is represented using a treemap of its package and file structure
- The size of a leaf node is proportional to the number of lines in the file it represents
- Example:

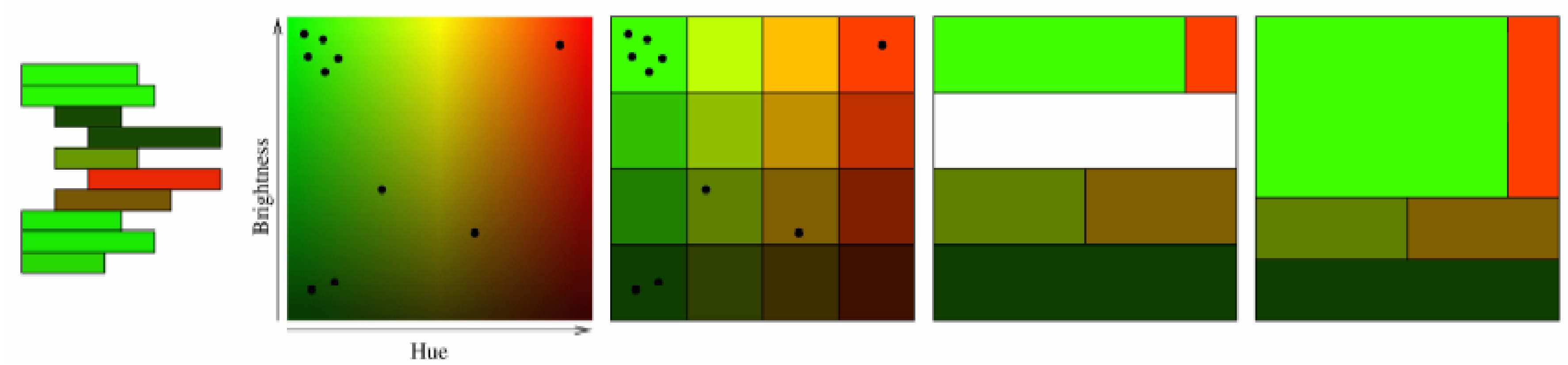


System Level Treemap

- Colour distribution of statements must be represented in the corresponding treemap node
- Nodes are coloured in proportion to the colours of lines in the corresponding file

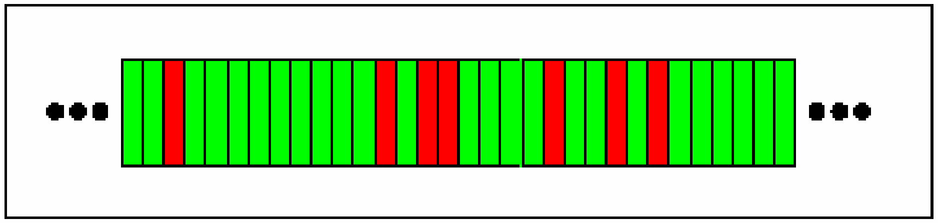
System Level Treemap

- File-node colouring algorithm

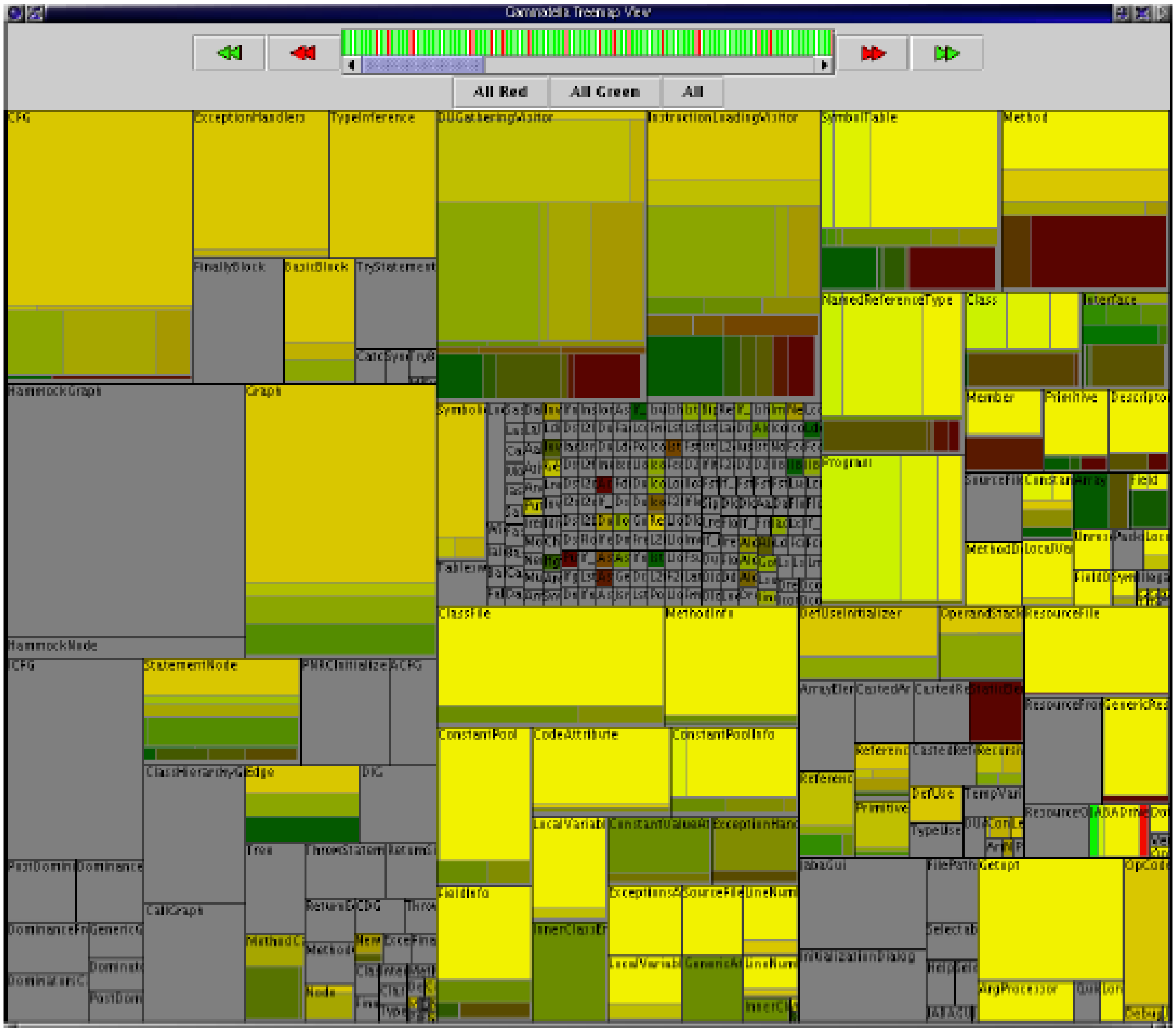


Miniature source-code view	Statement colours plotted on the hue-brightness space	Space is divided into discrete 'buckets'	Make the width of each 'bucket' proportional to the number of statements in the row	Make the height of each row proportional to the number of statements in the node

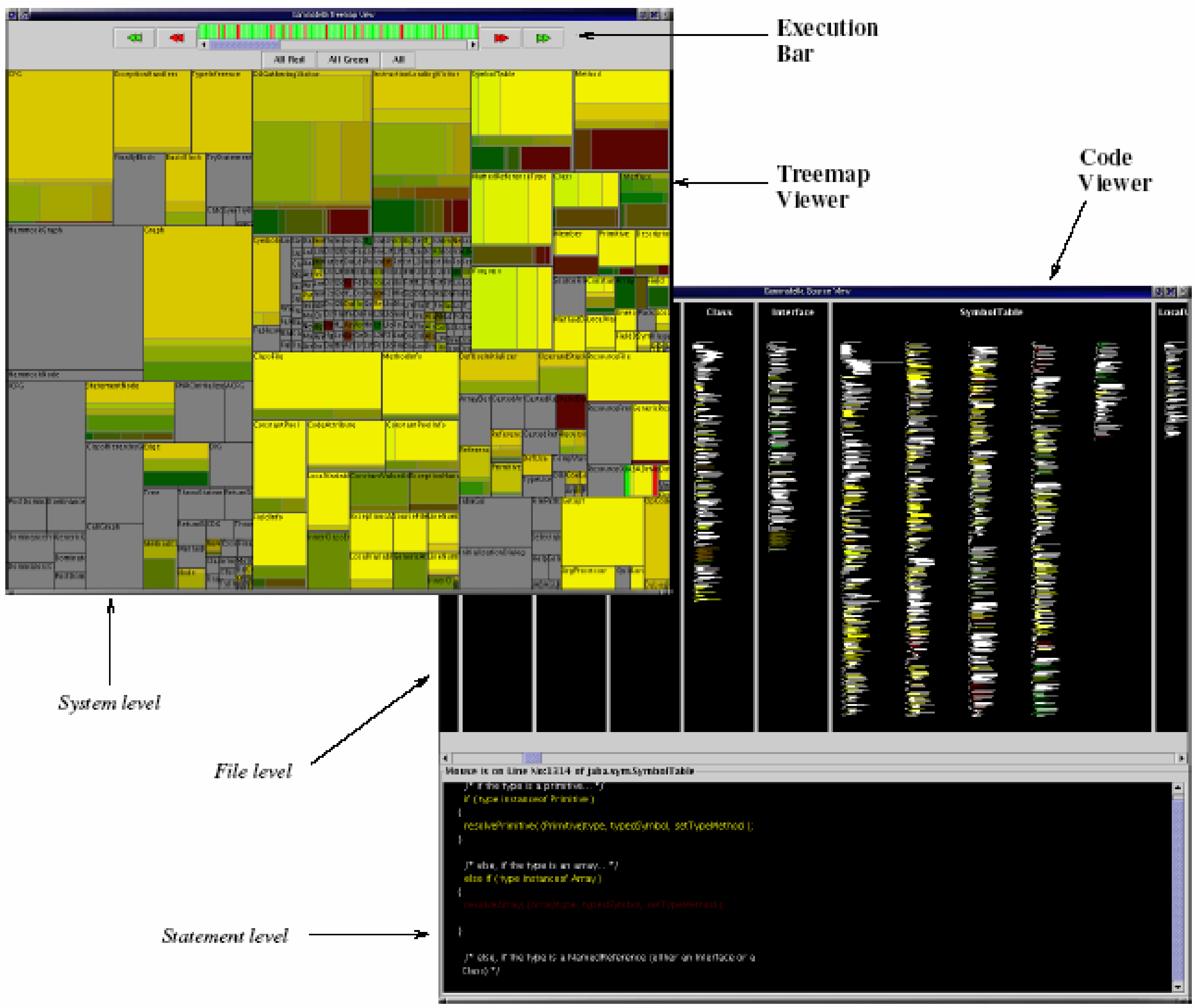
Execution Bar



- An “execution” represents a run of a program and the corresponding data collected
- Executions are represented as vertical bands on an *execution bar*
- Depending on the data being represented, hue or hue and brightness are used to determine the colour
- Scrollbars allow an unlimited number of executions to be displayed



Gammatella Treemap View
All Red
All Green
All
CFG
ExceptionHandlers
TypeInference
DUGatheringVisitor
InstructionLoadingVisitor
SymbolTable
Method
FinallyBlock
BasicBlock
TryStatement
NamedReferenceType
Class
Interface
HammockGraph
Graph
Member
Primitive
Descriptor
Program
SourceFile
ConstantArray
Field
MethodD
LocalVar
HammockNode
ClassFile
MethodInfo
DefUseInitializer
OperandStack
ResourceFile
ICFG
StatementNode
PNRCInitializeACFG
ArrayElem
CastedAr
CastedRe
ResourceFrom
GenericRes
ClassHierarchyGraph
Edge
DIG
ConstantPool
CodeAttribute
ConstantPoolInfo
Referenc
DefUse
TempVar
Primitive
TypeUse
ResourceO
LocalVariable
ConstantValueAt
ExceptionHandler
Tree
ThrowStatement
ReturnSt
PostDomin
Dominance
ReturnSDCDG
FieldInfo
ExceptionsA
SourceFile
LineNum
TabaGui
FilePath
Getopt
OpCode
CallGraph
DominanceFr
GenericG
MethodC
Method
New
ExceFina
InnerClassEr
LocalVariabl
GenericAt
LineNum
InitializationDialog
Selectab
HelpSelec
Dominate
DominatorC
PostDom
Node
InnerCl
ArgProcessor
Debug



```
Mouse is on Line No:1314 of jaba.sym.SymbolTable
  /* if the type is a primitive... */
  if ( type instanceof Primitive )
 {
  resolvePrimitive( (Primitive)type, typedSymbol, setTypeMethod );
 }

  /* else, if the type is an array... */
  else if ( type instanceof Array )
 {
  resolveArray( (Array)type, typedSymbol, setTypeMethod );

 }

  /* else, if the type is a NamedReference (either an interface or a
  Class) */
```

Filters and Summarizers

- Collected data is recorded as property-value pairs e.g. `java.version = 1.4.1_01`
- The executions visualized can be filtered using statements such as:

```
(java.version = '1.3.0') and
  (os.name = 'Linux')
```

- A 'summarizer' is a statement that instructs the system to aggregate executions with the specified properties

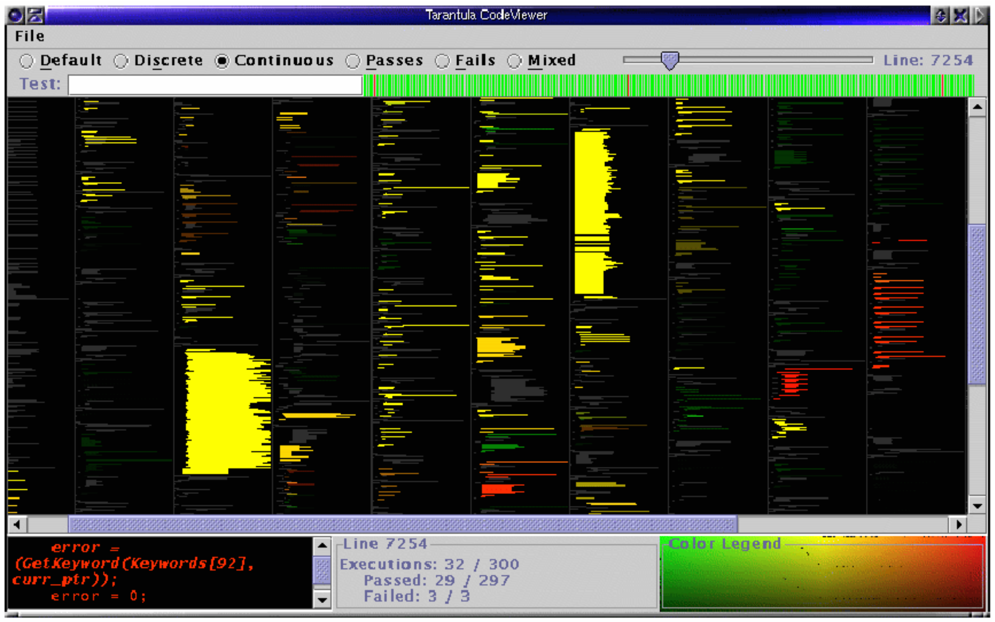
Feasibility Study

- Applied Gamma and Gammatella to JABA (Java Architecture for Bytecode Analysis)
- 550 Classes, 60KLOC
- Instrumentation caused a 28% reduction in performance
- Found many classes that were never used
- Found that JABA failed systematically when using the Sun JVM v. 1.4.0 on Solaris 2.8

Paper Critique

- Strengths
  - Scales to visualize larger systems than SeeSoft views alone
  - Solution can be generalized to many forms of analysis
  - Feasibility study suggests that valuable information can be gained from the system
- Weaknesses
  - Feasibility study suggests that instrumentation might be infeasible for many applications due to performance reduction
  - May be difficult to explore package structure – need to hover over package to get tool-tip with package name
  - Many file name labels are unreadable
  - Suggested colouring schemes for the execution bar were not explained
  - Colour mappings used in the feasibility study were not stated
  - Paper organization: Potential colour mappings not stated until the end

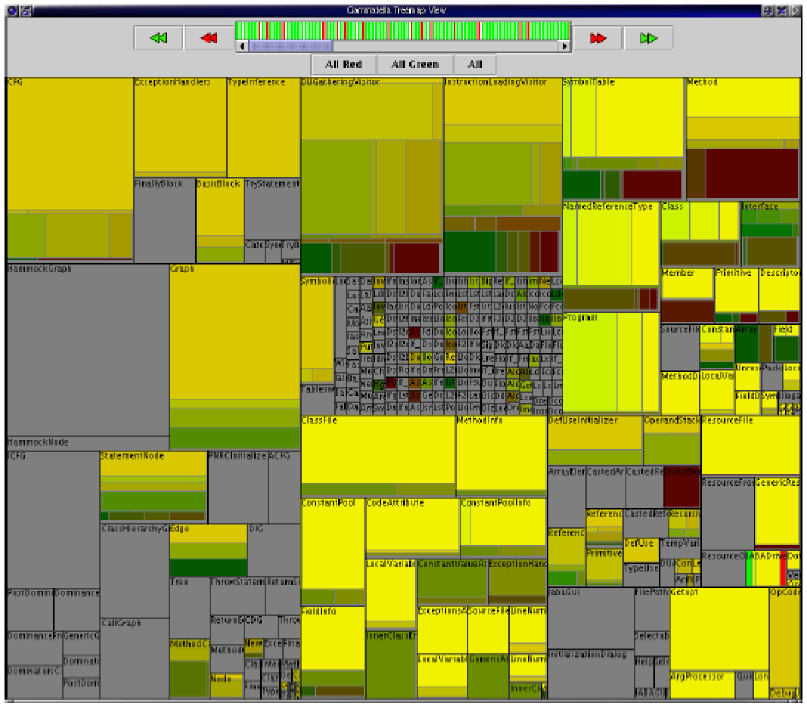
Questions?

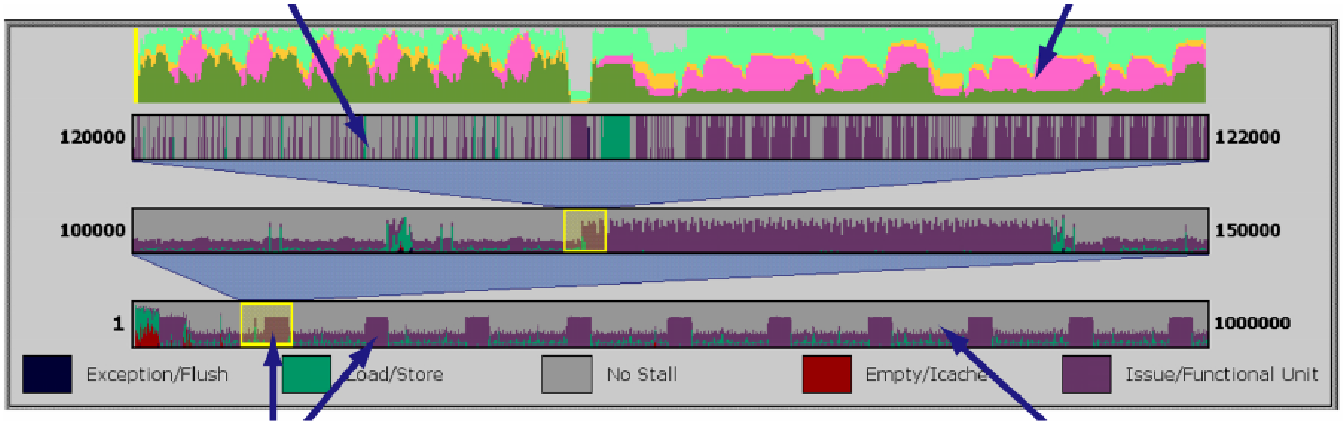


( ) Default ( ) Discrete (●) Continuous ( ) Passes ( ) Fails ( ) Mixed Line: 7254
Test:

```
error =
(GetKeyword(Keywords[92],
curr_ptr));
error = 0;
```

Line 7254
Executions: 32 / 300
Passed: 29 / 297
Failed: 3 / 3




Exception/Flush
Load/Store
No Stall
Empty/Icache
Issue/Functional Unit