Javelina:
An Advanced Code Coverage Package
Version 2.0

David R. “Chip” Kent IV
Los Alamos National Laboratory

October 6, 2005

Contents

1 Introduction **4**

2 Data Acquisition **4**
2.1 Javelina/Dyninst . . . 5
2.2 Javelina/Atom . . . 6

3 Data Manipulation **7**
3.1 Example 1: allblockstolines.py . . . 7
3.2 Example 2: allblockstolinesfast.py . . . 8
3.3 Example 3: oralllines.py . . . 9
3.4 Example 4: oralllinesfast.py . . . 9
3.5 Example 5: allblockstoreport.py . . . 10
3.6 Example 6: allblockstoreportfast.py . . . 10
3.7 Example 7: alllinestoreport.py . . . 11
3.8 Example 8: alllinestoreportfast.py . . . 11
3.9 Example 9: subtract2lines.py . . . 12

4 Data Visualization **13**

5 Javelina Python Application Program Interface (API) **15**
5.1 Modules and Packages . . . 15
5.1.1 ReportSorting . . . 15
5.2 Classes . . . 15
5.2.1 Coverage . . . 16
5.2.2 CoverageSummary . . . 18
5.2.3 ExecutableStructure . . . 19
5.2.4 ExecutionResult . . . 19
5.3 Functions . . . 19
5.3.1 AND . . . 19
5.3.2 AND_DESTROY . . . 20
5.3.3 NOT . . . 20
5.3.4 NOT_DESTROY . . . 20
5.3.5 OR . . . 21
5.3.6 OR_DESTROY . . . 21
5.3.7 REPORT . . . 21
5.3.8 SUBTRACT . . . 22

5.3.9 blockToLine 22
5.3.10 blockToLineFast 22
5.3.11 loadBlock 23
5.3.12 loadBlockFast 23
5.3.13 loadLine 23
5.3.14 loadStructure 24
5.3.15 summary 24

6 Installation 25
6.1 Data Acquisition 25
6.1.1 Javelina/Dyninst 25
6.1.2 Javelina/Atom 27
6.2 Manipulation/Visualization 27

7 Usage Notes (READ THIS!!) 28
7.1 Javelina/Dyninst 28
7.1.1 All Languages 28
7.2 Javelina/Atom 29
7.2.1 All Languages 29
7.2.2 C/C++ 29
7.2.3 Fortran 30
7.3 Javelina Python Libraries 31
7.4 Javelina GUI 31

8 Known Bugs 31
8.1 Javelina/Dyninst 31
8.1.1 All Languages 31
8.2 Javelina/Atom 32
8.2.1 Fortran 32
8.2.2 C 32
8.2.3 C++ 32
8.3 Javelina Python Libraries 33
8.4 Javelina GUI 33

9 Javelina Open Source Project 33

10 Help!! 34

A License 34

1 Introduction

Javelina is an advanced code coverage package. It allows users to:

- Acquire raw block-coverage data from an executable
- Convert block-coverage data to line-coverage data
- Perform advanced manipulations on line-coverage data
- Generate reports summarizing line-coverage results
- View source code with unexecuted lines highlighted

With Javelina, it is possible to combine coverage data from multiple executions and/or multiple processors during a single execution (in the cases of a parallel calculation). Using this data aggregation, the results of many test cases can be combined into a single result. Additionally, the results from many production-size parallel calculations can be aggregated into a single result showing what parts of the executable were used during actual calculations. This provides insight into what code is dead and can be removed.

Using Javelina's advanced functionality, it is possible to compare the lines executed in actual production calculations with the lines covered during tests. This can be very insightful in very large programs where only a small fraction of the code is ever executed during real runs.

The code-coverage viewer lets users quickly identify code which was never executed so that new test cases can be generated or the code can be classified as dead and removed.

For instructions in installing Javelina, see Section 6.

2 Data Acquisition

Javelina acquires coverage data using a variety of tools. Each tool supports a different set of platforms and has its own advantages and disadvantages. These tools are covered in the following subsections.

The Javelina/Dyninst data acquisition tool generates one type of file. This file indicates whether each analyzed line was or was not executed.

The Javelina/Atom data acquisition tool generates two types of files. One describes the structure of the binary and relates blocks to source-code line numbers. The other indicates which blocks were executed during a run. These two files can be combined into the same line coverage data as Javelina/Dyninst produces using the `blockToLine` function in the Javelina Python library.

2.1 Javelina/Dyninst

Javelina's Dyninst [3] based coverage tool inserts instructions into the binary being analyzed that record which blocks were executed during a run. Though Javelina/Dyninst records instruction-block coverage, only source-line coverage data is returned. This conversion is done on-the-fly as the coverage data is being collected and is equivalent to processing the data returned by Javelina's Atom tool using the `blockToLine` function.

All instrumentation done by Javelina's Dyninst tool is done using dynamic instrumentation. Dynamic instrumentation allows a binary's instructions to be edited after it is loaded into RAM. Javelina's Dyninst tool loads the analyzed binary into RAM and then inserts the appropriate instrumentation into the specified instruction blocks. After a block has been executed, its instrumentation is removed to minimize Javelina's overhead.

The first step in using Javelina's Dyninst tool is to compile the source code being analyzed with debugging information. The quality of Javelina's results depend on the quality of debugging information output by the compiler. For most compilers, adding "-g" to the command line will produce sufficient debugging information. Many compilers have other flags which can produce better quality debugging information. Many modern compilers can produce debugging information when optimization is used. See your compiler's documentation for further details. When using GCC [4] compilers, "-gdwarf-2" is reported to provide the best debugging information for Javelina's Dyninst tool.

Once an appropriate binary has been built, coverage data on the executable can be collected (`hello.x` in this example).

```
javelina hello.x
```

Running `javelina` produces a new file.

```
ls
    hello.c
    hello.x
    javelina.myhost.12345.xml
```

The new file contains the coverage data for `hello.x` execution on host myhost as PID 12345.

To collect data on specific shared libraries used by the executable, use the `--lib` command line argument.

```
javelina --lib libmylibrary.so hello.x
```

For a list of all possible command line arguments for Javelina's Dyninst tool, have the tool output its help page.

```
javelina --help
```

For MPI programs, coverage data is collected as in the following example for `hellompi.x`.

```
mpirun -np 4 javelina hellompi.x
```

Some MPI implementations have problems launching processes this way. See Section 8.1 for details.

2.2 Javelina/Atom

Javelina's Atom [1] tool inserts instructions into the binary being analyzed that record which blocks were executed during a run. It additionally generates a file indicating which source lines are associated with each block. If this file does not link blocks to source lines, recompile the program with debugging enabled.

The first step in using Javelina's Atom tool is to instrument the binary being analyzed (`hello.x` in this example).

```
atom hello.x -tool javelina
```

All of the statically loaded shared libraries used by the binary can also be analyzed simply by adding `-all` to the Atom command line. Running Atom produces a few new files.

```
ls
    hello.c
    hello.x
    hello.x.javelina
    javelina.structure.60a5d9c774e12d828dab9e60ed0f84366673af03.xml
```

`hello.x.javelina` is the instrumented binary, and `javelina.structure.<s>.xml` is the file relating the binary's blocks to source-code line numbers. `<s>` is the freely available SHA-1 cryptographic hash [2] (checksum) of the structure file. This aids in rapidly locating the structure file which goes with a particular executable.

Once atom has been run on the executable, the raw block-coverage data is generated by running the instrumented binary.

```
./hello.x.javelina
```

The block-coverage data is contained in the `javelina.block.<m>.<pid>.xml` files, where `<m>` is the machine name and `<pid>` is the process ID that generated the file. The `javelina.block.*.*.xml` and `javelina.structure.*.xml` files are used by Javelina's data manipulation tools.

3 Data Manipulation

Advanced code-coverage data manipulation is done using Python [8] scripts. Python is an interpreted, interactive, object-oriented programming language. It is often compared to Tcl, Perl, Scheme, or Java. Python provides numerous libraries and is portable to many architectures. Those not familiar with Python should consult a good reference such as:

- http://www.python.org
- *Learning Python* by Lutz and Ascher
- *Python Essential Reference* by Beazley
- *Python Pocket Reference* by Lutz

Javelina provides libraries to parse, save, and manipulate code-coverage data. This application program interface (API) is covered in Section 5. Example scripts can be found in the
`${JAVELINAROOT}/python-examples` directory.

3.1 Example 1: allblockstolines.py

This example script converts all of the block coverage data in the current directory into line coverage data. It assumes that all block files in the current directory use the same structure file, and that structure file is the only one in the directory. Because this example uses block and structure files, it only applies to Javelina/Atom.

```
from javelina import loadStructure, loadBlock, blockToLine
import string
import glob

blockFiles = glob.glob("javelina.block.*.xml")
structureFile = glob.glob("javelina.structure.*.xml")[0]

for blockFile in blockFiles:
    lines = blockToLine(structureFile,blockFile)
    lineFile = string.replace(blockFile,"block","line")
    lines.save(lineFile)
```

3.2 Example 2: allblockstolinesfast.py

This example script converts all of the block coverage data in the current directory into line coverage data. It assumes that all block files in the current directory use the same structure file, and that structure file is the only one in the directory. This version is optimized for speed. The API used for optimization may change in future versions of Javelina. Because this example uses block and structure files, it only applies to Javelina/Atom.

```
from javelina import loadStructure, loadBlockFast, blockToLineFast
import string
import glob

blockFiles = glob.glob("javelina.block.*.xml")
structureFile = glob.glob("javelina.structure.*.xml")[0]
structure = loadStructure(structureFile)

for blockFile in blockFiles:
    lines = blockToLineFast(structure,blockFile)
    lineFile = string.replace(blockFile,"block","line")
    lines.save(lineFile)
```

3.3 Example 3: oralllines.py

This example script calculates the logical OR of all the line coverage files in the current directory and saves the result as `orall.xml`.

```
from javelina import loadLine, OR
import string
import glob

lineFiles = glob.glob("javelina.line.*.xml")
outputFile = "orall.xml"

result = loadLine(lineFiles[0])

for lineFile in lineFiles:
    result = OR(result,lineFile)

result.save(outputFile)
```

3.4 Example 4: oralllinesfast.py

This example script calculates the logical OR of all the line coverage files in the current directory and saves the result as `orall.xml`. This version is optimized for speed. The API used for optimization may change in future versions of Javelina.

```
from javelina import loadLine, OR_DESTROY
import string
import glob

lineFiles = glob.glob("javelina.line.*.xml")
outputFile = "orall.xml"

result = loadLine(lineFiles[0])

for lineFile in lineFiles[1:]:
    result = OR_DESTROY(result,lineFile)

result.save(outputFile)
```

3.5 Example 5: allblockstoreport.py

The following example takes all of the block-coverage files from the current directory, combines them, and produces two reports. One report is sorted based upon the percent coverage, and the other is sorted based upon the number of unexecuted lines. Because this example uses block and structure files, it only applies to Javelina/Atom.

```
from javelina import *
import string
import glob

blockFiles = glob.glob("javelina.block.*.xml")
structureFile = glob.glob("javelina.structure.*.xml")[0]

data = blockToLine(structureFile,blockFiles[0])

for blockFile in blockFiles:
    line = blockToLine(structureFile,blockFile)
    data = OR(data,line)

REPORT(data,ReportSorting.line,"report.line.txt")
REPORT(data,ReportSorting.percent,"report.percent.txt")
```

3.6 Example 6: allblockstoreportfast.py

The following example takes all of the block-coverage files from the current directory, combines them, and produces two reports. One report is sorted based upon the percent coverage, and the other is sorted based upon the number of unexecuted lines. This version is optimized for speed. The API used for optimization may change in future versions of Javelina. Because this example uses block and structure files, it only applies to Javelina/Atom.

```
from javelina import *
import string
import glob

blockFiles = glob.glob("javelina.block.*.xml")
structureFile = glob.glob("javelina.structure.*.xml")[0]
```

```
structure = loadStructure( structureFile )

data = blockToLineFast(structure,blockFiles[0])

for blockFile in blockFiles[1:]:
    line = blockToLineFast(structure,blockFile)
    data = OR_DESTROY(data,line)

REPORT(data,ReportSorting.line,"report.line.txt")
REPORT(data,ReportSorting.percent,"report.percent.txt")
```

3.7 Example 7: alllinestoreport.py

The following example script aggrigates all of the line coverage data in the current directory using an OR operation. Summary reports of this data are then produced. One report is sorted by unexecuted lines and the other by percent coverage. It assumes that all block files in the current directory use the same structure file, and that structure file is the only one in the directory.

```
from javelina import loadLine, OR, REPORT, ReportSorting
import string
import glob

lineFiles = glob.glob("javelina.line.*.xml")

data = loadLine(lineFiles[0])

for lineFile in lineFiles:
    data = OR(data,lineFile)

REPORT(data,ReportSorting.line,"report.line.txt")
REPORT(data,ReportSorting.percent,"report.percent.txt")
```

3.8 Example 8: alllinestoreportfast.py

The following example script aggrigates all of the line coverage data in the current directory using an OR operation. Summary reports of this data are then produced. One report is sorted by unexecuted lines and the other by percent coverage. It assumes that all block files in the current directory use the same structure file, and that structure file is the only one in the directory. This version is optimized for speed. The API used for optimization may change in future versions of Javelina.

```
from javelina import loadLine, OR_DESTROY, REPORT, ReportSorting
import string
import glob

lineFiles = glob.glob("javelina.line.*.xml")

data = loadLine(lineFiles[0])

for lineFile in lineFiles[1:]:
    data = OR_DESTROY(data,lineFile)

REPORT(data,ReportSorting.line,"report.line.txt")
REPORT(data,ReportSorting.percent,"report.percent.txt")
```

3.9 Example 9: subtract2lines.py

The following example script applies the SUBTRACT operation to two line coverage files specified on the command line. The result is then saved, and reports are generated. The SUBTRACT operation extracts the lines executed in the first line coverage file, marks these lines as executed (unexecuted) if they are executed (unexecuted) in the second line coverage file, and returns the resulting line coverage object. This operator is useful in determining which lines executed by a user were tested. In this case, the first file would be the coverage data for the user applications, and the second file would be the coverage data for the test cases.

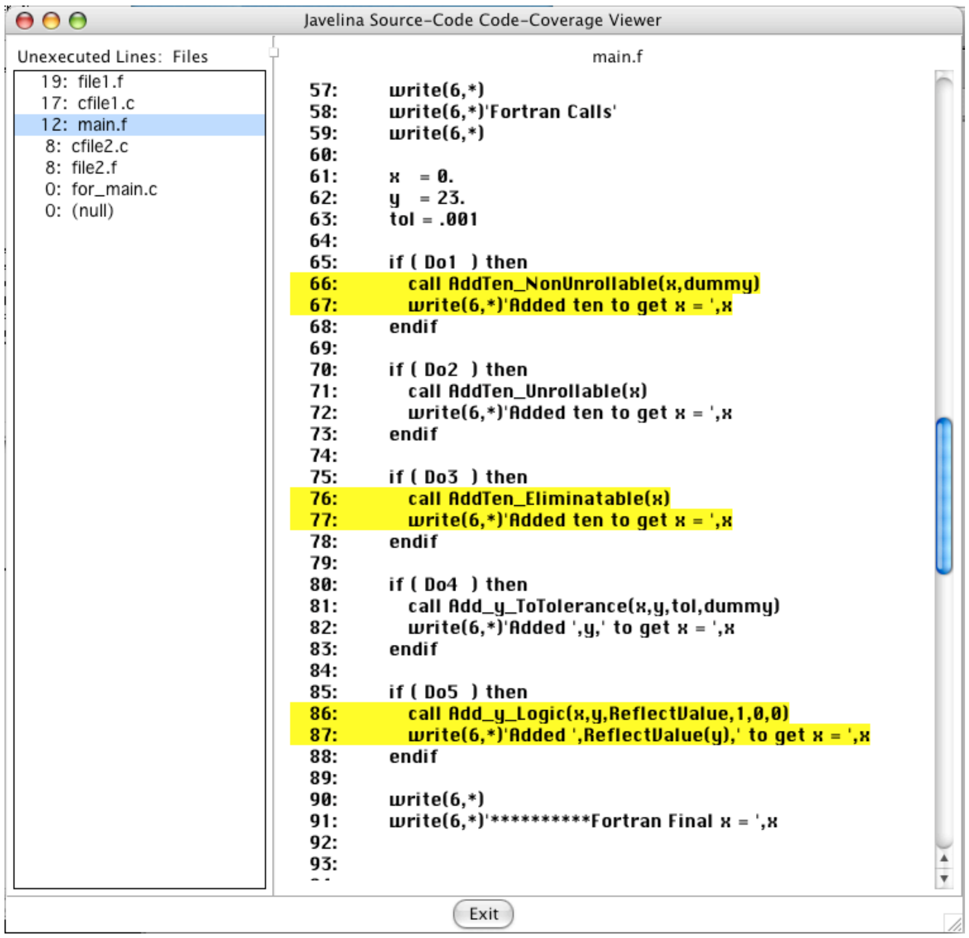
```
from javelina import SUBTRACT, REPORT, ReportSorting
import sys

if len(sys.argv) != 3:
    print "Usage: %s <userdata.xml> <testdata.xml>"%(sys.argv[0])
    sys.exit(1)
```

```
userdata = sys.argv[1]
testdata = sys.argv[2]

data = SUBTRACT(userdata,testdata)

data.save("subtract_result.xml")

REPORT(data,ReportSorting.line,"subtract_report.line.txt")
REPORT(data,ReportSorting.percent,"subtract_report.percent.txt")
```

4 Data Visualization

Code-coverage data and source code can be viewed using `javelinagui`. To use this tool, `cd` to the base directory where the source code to be analyzed is stored. When searching for a file, the name in the lines file will be opened if it exists. This could be an absolute path, a relative path, or a file name in the current directory. If the file does not exist in this location, all directories within the current directory will be searched for the file. Once in the source code directory, run `javelinagui` with a line coverage file as input.

```
cd /directory/with/source/code
javelinagui path/to/linecoveragedata.xml
```



```
19: file1.f
17: cfile1.c
12: main.f
 8: cfile2.c
 8: file2.f
 0: for_main.c
 0: (null)
```

```
      write(6,*)
      write(6,*)'Fortran Calls'
      write(6,*)

      x   = 0.
      y   = 23.
      tol = .001

      if ( Do1  ) then
        call AddTen_NonUnrollable(x,dummy)
        write(6,*)'Added ten to get x = ',x
      endif

      if ( Do2  ) then
        call AddTen_Unrollable(x)
        write(6,*)'Added ten to get x = ',x
      endif

      if ( Do3  ) then
        call AddTen_Eliminatable(x)
        write(6,*)'Added ten to get x = ',x
      endif

      if ( Do4  ) then
        call Add_y_ToTolerance(x,y,tol,dummy)
        write(6,*)'Added ',y,' to get x = ',x
      endif

      if ( Do5  ) then
        call Add_y_Logic(x,y,ReflectValue,1,0,0)
        write(6,*)'Added ',ReflectValue(y),' to get x = ',x
      endif

      write(6,*)
      write(6,*)'**********Fortran Final x = ',x
```

The GUI is very simple and intuitive to use. The files listed on the left are sorted by the number of lines marked as unexecuted in the line coverage input file. These lines are highlighted in yellow.

5 Javelina Python Application Program Interface (API)

Javelina provides a Python [8] module to load, save, and manipulate coverage data. This module can be accessed through either "`import javelina`" or "`from javelina import *`" in Python. This section describes the details of the application program interface (API).

In Python, it is possible to obtain this API documentation using the `help` function. For example:

```
import javelina
help(javelina.loadBlock)
```

5.1 Modules and Packages

This section covers the modules and packages available through the Javelina Application Program Interface (API).

5.1.1 ReportSorting

```
Functions used to sort coverage reports.

FUNCTIONS
    line(a, b)
        Sorting function based on the number of unexecuted
        lines.

    percent(a, b)
        Sorting function based on the percentage of unexecuted
        lines.
```

5.2 Classes

This section covers the classes available through the Javelina Application Program Interface (API).

5.2.1 Coverage

```
class Coverage
 |  Coverage information for a group of images.  An image is an
 |  executable or a shared library.
 |
 |  __init__(self, executableStructure=None, executionResult=None)
 |      Creates a new instance of this class.
 |
 |      executableStructure -- ExecutableStructure object
 |                             describing the executable
 |      executionResult     -- ExecutionResult object
 |                             describing which blocks were
 |                             executed
 |
 |  AND(self, other)
 |      Performs a logical AND operation on the data in two
 |      objects and returns the result.  A line will be marked
 |      as executed if both objects mark the line as having
 |      been executed.  AND does not modify its arguments.
 |      For a faster version of AND which destroys the
 |      arguments, see AND_DESTROY.
 |
 |      other  -- object to AND with this object
 |      return -- logical AND of the two objects
 |
 |  AND_DESTROY(self, other)
 |      Performs a logical AND operation on the data in two
 |      objects and returns the result.  A line will be marked
 |      as executed if both objects mark the line as having
 |      been executed.  AND_DESTROY modifies its arguments.
 |      For a slower version of AND_DESTROY which preserves
 |      the arguments, see AND.
 |
 |      other  -- object to AND with this object
 |      return -- logical AND of the two objects
 |
 |  NOT(self)
 |      Performs a logical NOT operation on the data in this
 |      object and returns the result.  A line will be marked
 |      as executed if it was not executed and vice versa.
 |      NOT does not modify its arguments.  For a faster
 |      version of NOT which destroys the arguments, see
 |      NOT_DESTROY.
 |
 |      return -- logical NOT of the this object
 |
 |  NOT_DESTROY(self)
 |      Performs a logical NOT operation on the data in this
 |      object and returns the result.  A line will be marked
 |      as executed if it was not executed and vice versa.
 |      NOT_DESTROY modifies its arguments.  For a slower
 |      version of NOT_DESTROY which preserves the arguments,
 |      see NOT.
 |
 |      return -- logical NOT of the this object
 |
 |  OR(self, other)
 |      Performs a logical OR operation on the data in two
 |      objects and returns the result.  A line will be marked
 |      as executed if either object marks the line as having
 |      been executed.  OR does not modify its arguments.  For
 |      a faster version of OR which destroys the arguments,
 |      see OR_DESTROY.
 |
 |      other  -- object to OR with this object
 |      return -- logical OR of the two objects
 |
 |  OR_DESTROY(self, other)
 |      Performs a logical OR operation on the data in two
 |      objects and returns the result.  A line will be marked
 |      as executed if either object marks the line as having
 |      been executed.  OR_DESTROY modifies the input
 |      arguments.  For a slower version of OR_DESTROY which
 |      preserves the arguments, see OR.
 |
 |      other  -- object to OR with this object
 |      return -- logical OR of the two objects
 |
 |  SUBTRACT(self, other)
 |      Extracts the lines of this object which have been
 |      executed, marks these lines as executed if they are
 |      executed in the other object, and returns the result.
 |      This operator is useful in determining which lines
 |      executed by a user were tested.
 |
 |      other  -- see description above
 |      return -- see description above
 |
 |  addImage(self, image)
 |      Adds an image to this object.
 |
 |      image -- image added to this object
 |
 |  save(self, fileName)
 |      Writes the coverage data out as an XML file.
 |
 |      fileName -- file where the data will be written
 |
 |  toXML(self, writer)
 |      Writes the coverage data out to a writer as XML.
 |
 |      writer -- writer object to write the data to
```

5.2.2 CoverageSummary

```
class CoverageSummary
 |  Summary of the coverage information contained in a
 |  Coverage object.
 |
 |  __init__(self, type, name)
 |      Creates a new instance of this class and initializes
 |      it.
 |
 |      type -- type of this section of code (e.g. file,
 |              procedure)
 |      name -- name for this section of code
```

5.2.3 ExecutableStructure

```
class ExecutableStructure
 |  Information on the structure of the analyzed executable.
 |
 |  __init__(self)
 |      Creates a new instance of this class.
```

5.2.4 ExecutionResult

```
class ExecutionResult
 |  Information on which blocks were and were not executed.
 |
 |  __init__(self, blockfile)
 |      Creates a new ExecutionResult.
 |
 |      blockfile -- signature of the static analysis file
 |                   which goes with this data.
```

5.3 Functions

This section covers the functions available through the Javelina Application Program Interface (API). These functions are used to load instances of the Javelina classes from files and are used to manipulate the data.

5.3.1 AND

```
AND(line1, line2)
    Performs a logical AND operation on two line covergae
    objects or files and returns the resulting line coverage
    object.  A line will be marked as executed if both object
    marks the line as having been executed.  AND does not
    modify its arguments.  For a faster version of AND which
    destroys the arguments, see AND_DESTROY.
    line1  -- line coverage object or data file
    line2  -- line coverage object or data file
    return -- line coverage object
```

5.3.2 AND_DESTROY

```
AND_DESTROY(line1, line2)
    Performs a logical AND operation on two line coverage
    objects or files and returns the resulting line coverage
    object.  A line will be marked as executed if both object
    mark the line as having been executed.  AND_DESTROY
    modifies its arguments.  For a slower version of
    AND_DESTROY which preserves the arguments, see AND.

    line1  -- line coverage object (not a data file)
    line2  -- line coverage object or data file
    return -- line coverage object
```

5.3.3 NOT

```
NOT(line)
    Performs a logical NOT operation on a line coverage
    object or file and returns the resulting line coverage
    object.  A line will be marked as executed if it was not
    executed and vice versa.  NOT does not modify its
    arguments.  For a faster version of NOT which destroys the
    arguments, see NOT_DESTROY.

    line  -- line coverage object or data file
    return -- line covearge object
```

5.3.4 NOT_DESTROY

```
NOT_DESTROY(line)
    Performs a logical NOT operation on a line coverage
    object and returns the resulting line coverage object.  A
    line will be marked as executed if it was not executed and
    vice versa.  NOT_DESTROY modifies its arguments.  For a
    slower version of NOT_DESTROY which preserves the
    arguments, see NOT.

    line  -- line coverage object (not a data file)
    return -- line coverage object
```

5.3.5 OR

```
OR(line1, line2)
    Performs a logical OR operation on two line coverage
    objects or files and returns the resulting line coverage
    object. A line will be marked as executed if either
    object marks the line as having been executed.  OR does
    not modify its arguments.  For a faster version of OR
    which destroys the arguments, see OR_DESTROY.

    line1  -- line coverage object or data file
    line2  -- line coverage object or data file
    return -- line coverage object
```

5.3.6 OR_DESTROY

```
OR_DESTROY(line1, line2)
    Performs a logical OR operation on two line coverage
    objects or files and returns the resulting line coverage
    object. A line will be marked as executed if either object
    marks the line as having been executed.  OR_DESTROY
    modifies the input arguments.  For a slower version of
    OR_DESTROY which preserves the arguments, see OR.

    line1  -- line coverage object  (not a data file)
    line2  -- line coverage object or data file
    return -- line coverage object
```

5.3.7 REPORT

```
REPORT(data, sorting, outfile)
    Generates a coverage summary report from either a
    summary data object, a line coverage object, or a file
    containing line coverage data.  The report is sorted
    according to the given criteria.

    data    -- a summary data object, a line coverage object,
               or a file containing line coverage data
    sorting -- a sorting function
    outfile -- output file where the report will be stored
```

5.3.8 SUBTRACT

```
SUBTRACT(line1, line2)
    Extracts the lines of the line1 line coverage object or
    data file which have been executed, marks these lines as
    executed if they are executed in the line2 line coverage
    object or datafile, and returns the resulting line
    coverage object.  This operator is useful in determining
    which lines executed by a user were tested.

    line1  -- line coverage object or data file
    line2  -- line coverage object or data file
    return -- line coverage object
```

5.3.9 blockToLine

```
blockToLine(structure, block)
    Takes a structure object or datafile and a block
    coverage object or datafile and returns a line coverage
    object.  This function uses a strict XML parser so it very
    robust, though slow.  For a fast and less robust parser,
    see blockToLineFast.

    structure -- structure object or datafile
    block     -- block coverage object or datafile
    return    -- line coverage object
```

5.3.10 blockToLineFast

```
blockToLineFast(structure, block)
    Takes a structure object or datafile and a block coverage
    object or datafile and returns a line coverage object.
    This function does not use a strict XML parser so it may
    be confused if the data file output by Javelina has been
    modified or is from a different version of Javelina.  This
    is a tradeoff for faster parsing speeds.  For a strict,
    safe, and slow parser, use blockToLine.  For a fast and
    less robust parser, use blockToLineFast.

    structure -- structure object or datafile
    block     -- block coverage object or datafile
    return    -- line coverage object
```

5.3.11 loadBlock

```
loadBlock(file)
    Loads the block coverage for an execution from an XML
    file.  This function is a strict XML parser so it very
    robust, though slow.  For a fast and less robust parser,
    see loadBlockFast.

    file   -- file containing the block coverage data
    return -- block coverage information for the execution
```

5.3.12 loadBlockFast

```
loadBlockFast(file)
    Loads the block coverage for an execution from an XML file.
    This function is not strict XML parser so it may be confused
    if the data file output by Javelina has been modified or is
    from a different version of Javelina.  This is a tradeoff for
    faster parsing speeds.  For a strict, safe, and slow parser,
    use loadBlock.  For a fast and less robust parser, use
    loadBlockFast.

    file   -- file containing the block coverage data
    return -- block coverage information for the execution
```

5.3.13 loadLine

```
loadLine(file)
```

```
    Loads line coverage data from an XML file.

    file   -- file containing the line coverage data
    return -- line coverage data
```

5.3.14 loadStructure

```
loadStructure(file)
    Loads the structure of an executable from an XML file.

    file   -- file containing the executable structure
    return -- structure of the executable
```

5.3.15 summary

```
summary(coverage)
    Generates a coverage summary for the given line coverage
    object or data file.

    coverage -- line coverage object or data file.
```

6 Installation

Once the Javelina distribution file has been downloaded, unzip it into the directory where it will be installed.

```
cp javelina-<version>.tar.gz /directory/to/install/javelina
cd /directory/to/install/javelina
tar -zxvf javelina-<version>.tar.gz
```

Now that Javelina is installed, the `JAVELINAROOT` environment variable must be set:

```
setenv JAVELINAROOT /directory/to/install/javelina
```

or

```
export JAVELINAROOT=/directory/to/install/javelina
```

Once `JAVELINAROOT` is set, follow the instructions in the following subsections to complete the setup. To ease setup, `sourceme.csh` and `sourceme.sh` are provided. Edit these files and then source them to setup the proper environment variables for Javelina. For csh or tcsh use:

```
source ${JAVELINAROOT}/sourceme.csh
```

and for bash use:

```
source ${JAVELINAROOT}/sourceme.sh
```

6.1 Data Acquisition

Javelina's data acquisition functionality can be used without installing the manipulation/visualization functionality. This is useful when data is acquired on one machine and analyzed on another.

6.1.1 Javelina/Dyninst

Javelina/Dyninst data acquisition tool is built on the University of Maryland and the University of Wisconsin's Dyninst [3] dynamic instrumentation library (version $\geq$ 5.0). Dyninst and Javelina/Dyninst are portable to most common computing environments. See the Dyninst web site for a current list of supported platforms. Dyninst source code and binaries can be downloaded from http://www.dyninst.org and used for free within an organization. Additionally, Javelina/Dyninst requires the Scons [9] build system (http://www.scons.org) as well as the Boost [10] (version $\geq$ 1.33) C++ libraries (http://www.boost.org).

Once Dyninst, Scons, and Boost have been installed, a few environment variables must be set:

PLATFORM Specification of what specific architecture and operating system is being used. For x86/Linux, this has a value of `i386-unknown-linux-2.4`, and for x86_64/Linux, this has a value of `x86_64-unknown-linux2.4`. Values for other platforms can be found by looking in the `core/dyninstAPI` directory of the Dyninst source code.

BOOSTROOT Directory where Boost is installed. This directory contains Boost's `include` and `lib` directories.

BOOSTVERSION Specification of what version of Boost is being used. For example, if Boost version 1.33 is used, this value should be set to `boost-1_33`.

DYNINSTINCLUDE Directory containing the Dyninst include files. When building Dyninst from source, this directory will be `core/dyninstAPI/h` under the Dyninst base directory.

DYNINSTROOT Directory where Dyninst is installed. This directory contains Dyninst's `bin` and `lib` directories. When building Dyninst from source, this directory will be `${PLATFORM}` under the Dyninst base directory.

DWARFROOT Directory where Libdwarf is installed. This directory contains Libdwarf's `include` and `lib` directories. This variable is only necessary when using Dyninst on a platform which requires the Libdwarf libraries.

PATH The search path for executables. To add Javelina/Dyninst to the standard search path, set `PATH` to `${JAVELINAROOT}/bin/${PLATFORM}/${BUILDTYPE}:${PATH}` where `${BUILDTYPE}` is the type of Javelina build done (see below).

LD_LIBRARY_PATH The search path for shared libraries. To add the Javelina/Dyninst libraries to the standard search path, set `LD_LIBRARY_PATH` to `${JAVELINAROOT}/lib/${PLATFORM}${BUILDTYPE}:${PATH}` where `${BUILDTYPE}` is the type of Javelina build done (see below). Some platforms use different names for `LD_LIBRARY_PATH` when either 32- or 64-bit software can be executed by the machine. Use the appropriate environment variable in these situations (e.g. `LD_LIBRARY64_PATH` on IRIX). Linux machines use `LD_LIBRARY_PATH` for such machines.

After the appropriate environment variables are set, Javelina/Dyninst can be built. Simply "`cd ${JAVELINAROOT}`", and run "`scons`". This builds the debug version of Javelina by default. An optimized version of Javelina can be created using "`scons release`". Now
Javelina/Dyninst is ready to use.

6.1.2 Javelina/Atom

The Javelina/Atom data acquisition tool is built on HP's Atom tool. Atom [1] is proprietary and runs only on the Tru64 OS.

To enable Atom based data acquisition, the `ATOMTOOLPATH` environment variable must be set:

```
setenv ATOMTOOLPATH ${JAVELINAROOT}/atom:${ATOMTOOLPATH}
```

or

```
export ATOMTOOLPATH=${JAVELINAROOT}/atom:${ATOMTOOLPATH}
```

6.2 Manipulation/Visualization

Javelina's manipulation/visualization functionality can be used without installing the data acquisition functionality. This is useful when data is acquired on one machine and analyzed on another.

All of Javelina's data manipulation/visualization functionality is written in Python. Python version $\geq$ 2.2 with a SAX XML parser must be installed. Essentially all recent Python installations satisfy this requirement. If a `SAXReaderNotAvailable` exception is raised while using Javelina, then a Python SAX XML parser is not installed. A new version of the Python interpreter, which contains this parser, can be downloaded [8] and installed.

To use the source-code browser, Python's Tkinter module must be installed. If the module is not installed, Python will raise an exception stating `ImportError: No module named _tkinter` when trying to run the browser. Tkinter is Python's official GUI library so it is present in almost all installations. If Tcl/Tk [11] is installed in a standard location, the Tkinter module will automatically be created when a new version of the Python interpreter is built.

Python with a SAX parser and Tkinter is available on Linux, Unix, Mac OS X, Windows, and probably a few more platforms.

To use Javelina's data manipulation/visualization functionality, the `PYTHONPATH` and `PATH` environment variables must be set.

```
setenv PYTHONPATH ${JAVELINAROOT}/python:${PYTHONPATH}
setenv PATH ${JAVELINAROOT}/bin:${PATH}
```

or

```
export PYTHONPATH=${JAVELINAROOT}/python:${PYTHONPATH}
export PATH=${JAVELINAROOT}/bin:${PATH}
```

7 Usage Notes (READ THIS!!)

The section contains notes on using Javelina that every user should read (especially before submitting a bug report). They should be kept in mind when interpreting data from Javelina.

7.1 Javelina/Dyninst

7.1.1 All Languages

- Javelina/Dyninst maps instructions to source lines using the debugging information provided by the compiler. Depending on the debugging information format, the compiler's optimization level, the particular brand compiler, and the version of binutils on the system, the quality of the line numbers will vary. In general, using no optimizations and the DWARF2 debugging format will provide the highest quality data. Experimenting with optimization levels will show how much "fuzzing" of the coverage information happens. Choose an appropriate balance of execution speed and coverage quality for your particular application.

For the GCC compilers [4], using "`-O0 -gdwarf-2`" seems to provide the highest quality data on Linux systems.

- Testing for Javelina/Dyninst has been most extensive on x86/Linux and x86_64/Linux systems using the GCC compilers. Expect these combinations to have the fewest issues.
- Testing for Javelina/Dyninst has been most extensive with Open-MPI [5]. Other MPI implementation may work, but they haven't been tested. See Section 8.1 for more information.
- Javelina/Dyninst does not follow the children of `fork` calls. This can cause problems in examining the coverage of code that forks. Also, MPI implementations that use `fork` to spawn processes (e.g. LA-MPI [6]) likely will not function properly.
- Javelina version 2.0 can not analyze multi-threaded software. This is a result of a Dyninst limitation. Multi-threaded support will be added to Javelina as soon as Dyninst can cope with threads.

7.2 Javelina/Atom

7.2.1 All Languages

- In a source file, a single statement can extend to multiple lines. In a symbol table, it is common for the compiler to associate all lines of a multi-line statement with only the first source line. For example, the statement

```
int i = a + b
          + c
          + d;
```

  may all be assigned to the first line of the statement.

7.2.2 C/C++

- In most contexts "{" and "}" are not executable statements. They typically do nothing. For example, the "{" and "}" in

```
int main()
{
return 1;
}
```

do nothing.

- Statements such as `break;` and `return;` may be removed during optimization. Because of this, these statements may not be executable statements in the binary.
- The Alpha cxx compiler does not do a good job of generating symbol tables when using a high level of optimization. If an optimized binary is giving unusual data, try turning down the optimization or using a different compiler.
- When using declaring an STL `string` (e.g. `string temp;`) anywhere in a file, there is a bug in the symbol table generated by the Alpha cxx compiler. The bug causes line 230 to be marked as executed even when the line is clearly not executable (e.g. whitespace, comment, etc.). Just ignore line 230 if you see strange results. This will likely get moved from bugs to notes with further testing.

7.2.3 Fortran

- Statements such as `else` and `endif` are not executable statements in most contexts. They simply denote the beginning or end of a code block.
- `format` statements are sometimes marked as being executable and are sometimes not. This is an inconsistency in the Alpha Fortran compilers symbol table generation. In addition, `format` statements which are marked as executable by the compiler are moved to the start of the function they appear in during optimization. No matter where a `format` statement appears in a function, it will *always* be marked as executed if the function is executed. Because of these issues, `format` statements should be ignored when viewing coverage data.

- `write` statements which contain formatting information within them are split by the Alpha Fortran compilers into a `write` statement and a `format` statement. The `format` portion of the statement will always be executed if the function is executed (see above). Because of this, `write` statements containing formatting information will *always* be marked as executed if the function containing them is called. Because of these issues, `write` statements should be ignored when viewing coverage data.

7.3 Javelina Python Libraries

There are currently no usage notes for the Javelina python libraries.

7.4 Javelina GUI

There are currently no usage notes for the Javelina GUI.

8 Known Bugs

Unfortunately Javelina does have a few bugs. If any new bugs are found, please report them to http://javelina.tigris.org or javelina@lanl.gov.

8.1 Javelina/Dyninst

8.1.1 All Languages

- If Javelina/Dyninst is used on a binary which has no debugging information, Javelina/Dyninst may segfault. This is a known problem with Dyninst which is being corrected.

- Javelina version 2.0 can not analyze multi-threaded software. This is a result of a Dyninst limitation. Multi-threaded support will be added to Javelina as soon as Dyninst can cope with threads.

- The prerelease versions of Dyninst 5.0 are *extremely* slow when associating line numbers with instructions. Performance is ok when analyzing 10 MB executables, but it has been intolerably slow for 150+ MB executables.

- Javelina/Dyninst does not work with LA-MPI [6]. LA-MPI has an unusual job launching system that uses `fork` to spawn MPI processes. Since Javelina/Dyninst does not follow the children of forked processes, Javelina/Dyninst does not collect coverage information on the MPI processes. Development of LA-MPI has stopped, and it is being replaced by Open-MPI [5]. LA-MPI users should switch to Open-MPI to use Javelina.

- Javelina/Dyninst does not work with MPICH-P4 [7]. During job launching, MPICH-P4 appends arguments to the javelina command line. Javelina/Dyninst does not recognize these extra arguments and shuts down. It would be possible to have Javelina/Dyninst ignore these arguments. This will be implemented when there is enough demand to justify the change. MPICH-MPD [7] probably will not suffer from this problem, but it has not been tested. Open-MPI [5] is the most tested and recommended MPI implementation.

8.2 Javelina/Atom

8.2.1 Fortran

There are no reported bugs for using Javelina/Atom on binaries created with Fortran. Only the native Tru64 compilers have been tested.

8.2.2 C

There are no reported bugs for using Javelina/Atom on binaries created with C. Only the native Tru64 compilers have been tested.

8.2.3 C++

- When Atom is used on an executable generated with g++, every instruction is assigned to either source line 0 or 6. This is an Atom bug. Since HP has discontinued the Tru64 platform, this bug will never be fixed. Use the Dyninst version of Javelina if this is an issue.

- Templates are listed once for each translation unit (source file with all necessary files included) they are used in. Every file with includes `template.h` will produce its own data for `template.h`. The data from all translation units needs to be combined into one result, which is associated with the original header file.

- If the compiler breaks a function into multiple functions during optimization, the data indicating if a line was or was not executed is not always correct in the GUI. This is a rare event in practice.

- The functions `atof`, `atoi`, and `exit` will often be marked as executed when they clearly are not. This may be an issue with the symbol table, preprocessor macros, or inlining. This is seen with the Alpha cxx compiler.

- When using the Alpha cxx compiler, `if(...){...}` statements can behave strangely. In some cases the if statement will be marked as unexecuted and the statements in the {...} will be marked as executed. Other times some of the statements in the {...} will be marked as executed and others won't. Other times, the if part of the statement clearly had to be executed but is marked as unexecuted. These problems have been observed when strings are used in the if statement or {...}. These problems have been seen with the Alpha cxx compiler. Please report any instances of this so that the cause can be identified.

8.3 Javelina Python Libraries

The Javelina python libraries have no reported bugs.

8.4 Javelina GUI

The Javelina GUI has no reported bugs.

9 Javelina Open Source Project

The Javelina open source project [12] is hosted at http://javelina.tigris.org. This is the source for the latest releases of Javelina and the most up-to-date information. Submit all bug reports, bug fixes, and feature requests at this site. There are also discussion lists where users can obtain help.

10 Help!!

For additional help with Javelina, see the Javelina open source project [12] at http://javelina.tigris.org.

A License

Developed by David R. “Chip” Kent IV at Los Alamos National Laboratory.

Javelina is unclassified and released under LA-CC-04-128.

References

[1] Atom, http://h30097.www3.hp.com/dcpi/part1/sld011.htm

[2] SHA-1, http://www.itl.nist.gov/fipspubs/fip180-1.htm

[3] Dyninst, http://www.dyninst.org

[4] GCC, http://gcc.gnu.org

[5] Open-MPI, http://www.open-mpi.org

[6] LA-MPI, http://public.lanl.gov/lampi/

[7] MPICH, http://www-unix.mcs.anl.gov/mpi/mpich/

[8] Python, http://www.python.org

[9] Scons, http://www.scons.org

[10] Boost, http://www.boost.org

[11] TCL, http://tcl.sourceforge.net/

[12] Javelina Open Source Site, http://javelina.tigris.org