Python Profiling and Visualization

Richard Saunders *rts@rincon.com*
(Rincon Research Corporation)

Clinton Jeffery, Michael Wilder
{jeffery,mwilder}@cs.nmsu.edu
(New Mexico State University)

This work has graciously been sponsored by Rincon Research Corporation (RRC) and is copyright RRC 2004/2005,
but most work has been open sourced.

Background

- Rincon Research Corporation: We build Digital Signal Processing (DSP) Applications
  - DSP applications are VERY compute-intensive
    - Fast Fourier Transforms (FFT), filters, demodulation, etc.
- Applications built using proprietary MIDAS
  - Component-based: Compute-intensive components written in C/C++/FORTRAN
  - Script-based: Components assembled/connected with scripting (glue) language

What We Want: The Ideal

- RRC Problem: Building DSP applications is HARD
  - Need performance of C/C++/FORTRAN but the flexibility of Python (see “An empirical comparison of C, C++, Perl, Python, Rexx, and Tcl”, IEEE Computer)
- Recall “Uncle” Don Knuth's Maxim
  - 95% of run-time spent in 5% of code
- *Ideal Solution*
  - First, write 100% of application in Python
  - Profile to find hot spots and rewrite that 5% in C/C++

Current Python Profiling Tools

- *Ideal Solution* assumes an abundance of profiling tools, but not as many as we'd like. Currently:
  - Python has two run-to-completion profilers
    - `profile` module: Written in Python. Easy to read!, but runs slowly, doesn't profile C routines of C Python Modules (supposedly fixed in Python 2.4)
    - `hotspot` module: Written in C. Harder to read, runs faster, but postings on newsgroups don't give glowing reviews
  - Both use the *profiling hooks* already in Python
    - Deterministic profiling: catches every function call, return, exception

Approach

- RRC needs *steady state debuggable* applications
    - While the program is running, we can debug it
  - Site tunable: applications run in environments where they need to be profiled/tuned where installed
  - Dynamic profiling: turn on/off while running
  - Minimal intrusion: cheap enough for production code
- Two-prong approach:
  - `top` for Python: watch profile of program as it runs
  - Visualization tools: watch time spent in Python VM

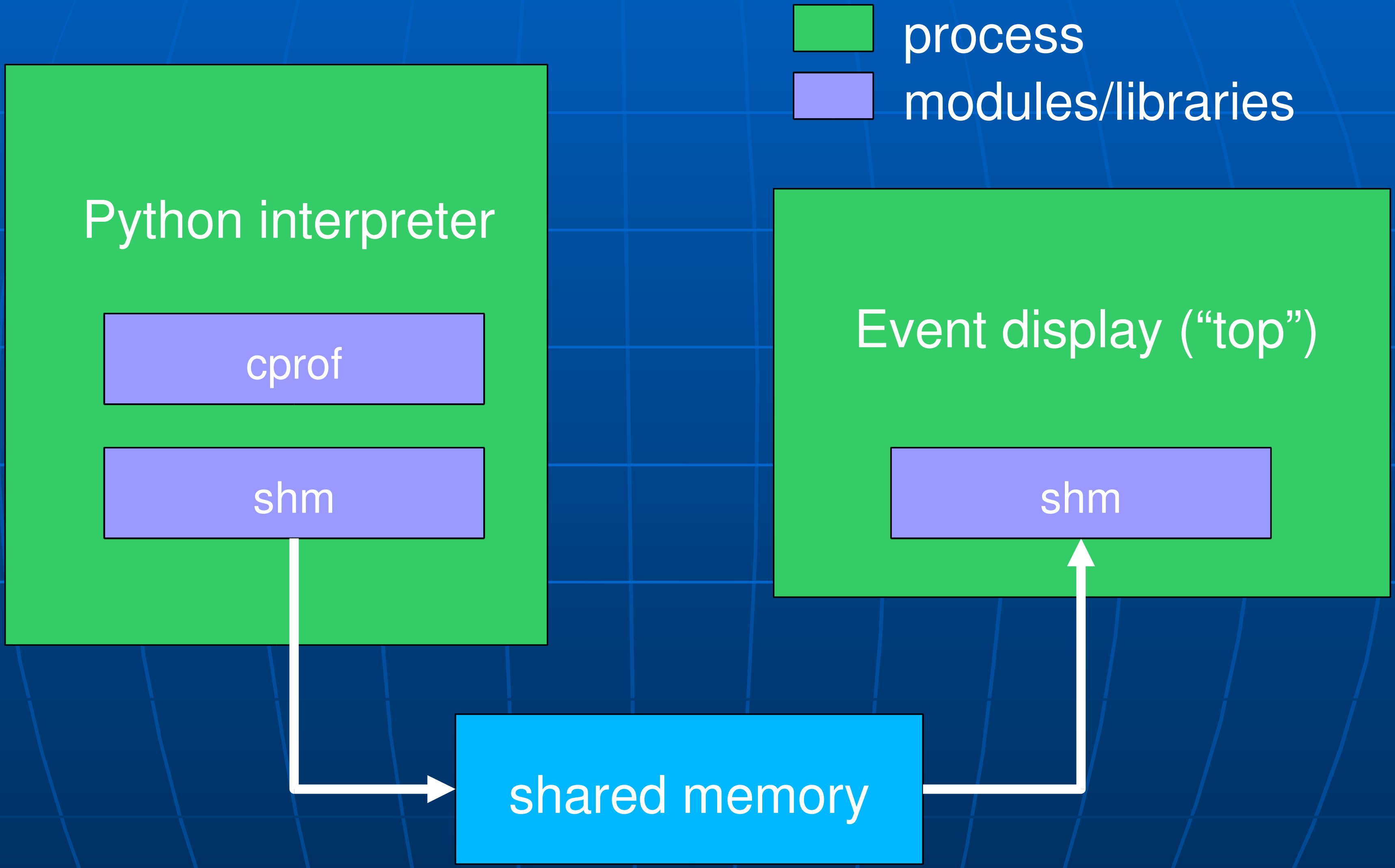
Python Top: Example

```
 ============================================================================
**FUNCTION NAME**:*TOTAL TIME****%*:*%DESC:*%PROC:*FILENAME*************:*LINE#
____________________________________________________________________________
:writestr        : 1.9500e+10:97.7 : 00.0 : 99.9 :         pyText2Pdf.py:  340
:StartPage       : 3.1646e+08:01.6 : 74.9 : 25.0 :         pyText2Pdf.py:  454
:EndPage         : 1.2225e+08:00.6 : 63.1 : 36.8 :         pyText2Pdf.py:  505
:WriteHeader     : 1.6958e+06:00.0 : 72.8 : 27.1 :         pyText2Pdf.py:  398
:parseArgs       : 2.1563e+05:00.0 : 26.9 : 73.0 :         pyText2Pdf.py:  306
:?               : 1.0504e+05:00.0 : 10.6 : 89.3 : b/python2.3/getopt.py:   16
:__init__        : 5.3041e+04:00.0 : 00.0 : 99.9 :         pyText2Pdf.py:  176
:getopt          : 3.8645e+04:00.0 : 00.0 : 99.9 : b/python2.3/getopt.py:   52
:pyText2Pdf      : 3.2025e+04:00.0 : 00.0 : 99.9 :         pyText2Pdf.py:  174
:argsCallBack    : 1.9461e+04:00.0 : 00.0 : 99.9 :         pyText2Pdf.py:  221
:GetoptError     : 1.1105e+04:00.0 : 00.0 : 99.9 : b/python2.3/getopt.py:   39
============================================================================
```

Python top: Implementation

- Uses deterministic profiling
  - Catches all function calls and returns events via built-in hook in Python (uses C hook for speed)
- Timestamps each event
  - Super cheap: uses single `rttscll` instruction on Intel (cycle counter)
- Uses ULMA (Ultra Lightweight Monitoring Architecture)
  - Sends event to “Python top” in another CPU
    - Avoids computing “top” information in same CPU as running program
- Techniques applicable to other languages (C++ example)

Python top Diagram



🟩 process

🟪 modules/libraries

Python interpreter

cprof

shm

Event display (“top”)

shm

shared memory

Alamo: A Monitoring Framework

- Alamo (work of Dr. Jeffery)
  - Has 118+ events for VM and runtime system events
  - Useful for writing event-driven visualizations
  - Written in Unicon, a high-level language (from unicon.org, also on sourceforge) similar to Python
  - Alamo tends to have tools for Virtual Machine events
    - list, string, tables, etc. X creation, destruction, access
  - Covered in the book: *Program Monitoring and Visualization: An Exploratory Approach*

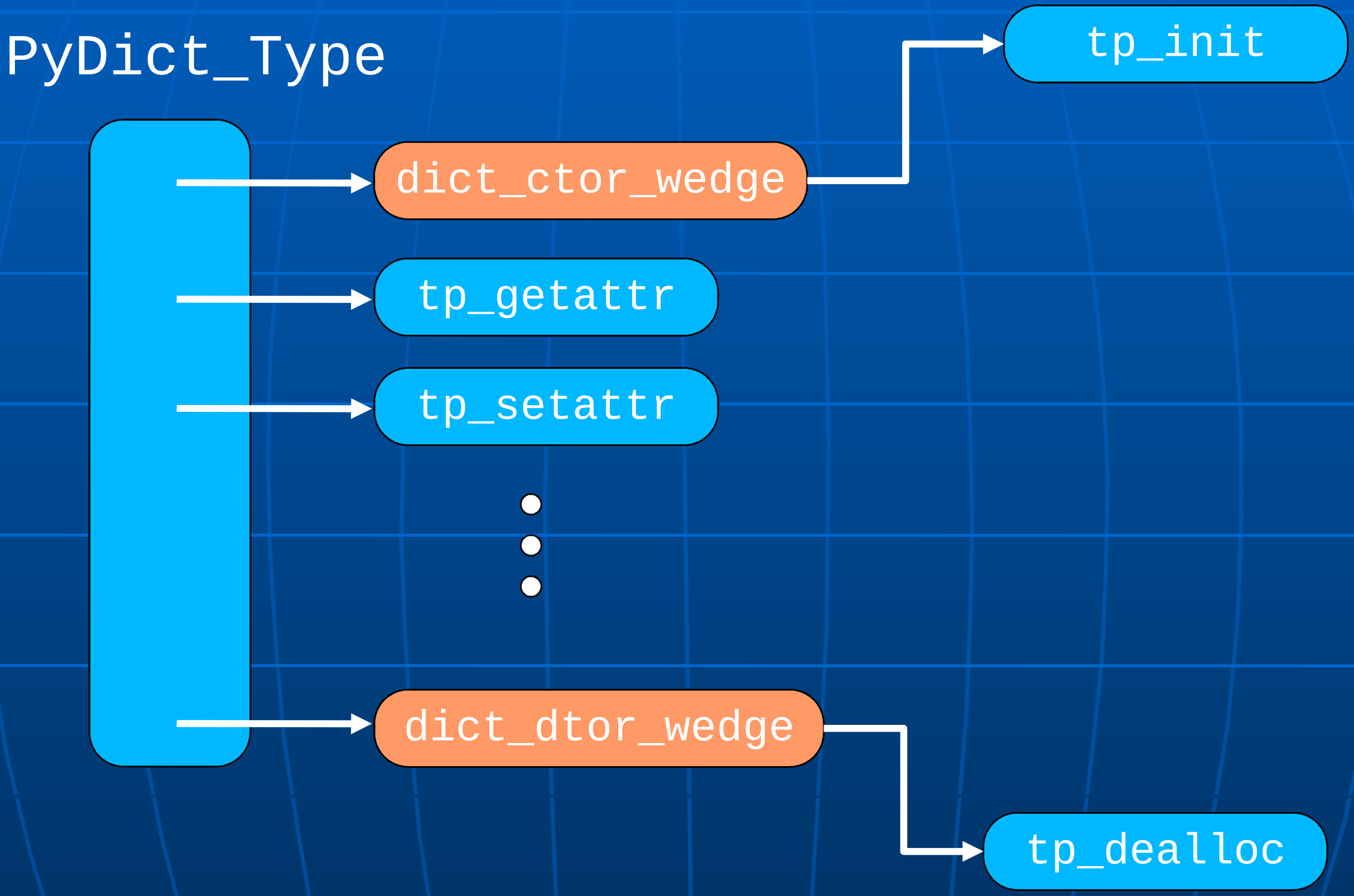
ULMA

- Ultra Lightweight Monitoring Architecture
  - How fast can we send events?
  - Approximation for Alamo's rich event set for Python, but more flexible in terms of event handling and communication mechanisms
- On-going work, hoping to write a paper
  - How fast can we send events in same process? Machine? Network?
    - Lightweight Events: Can send pointers, very cheap
    - Heavyweight Events: Have to do “deep copy” of info

Adding Hooks to Python

- Events:
  - Creation: typically can intercept “Meta” Objects construction events at run-time: wedge in to turn on, wedge out
    - *No need to change any Python VM code!*
    - *There is no extra overhead if not instrumenting!*
    - (Sometimes, have to change code in Python/Objects: in `listobject.c: PyListNew` also creates objects)
  - Deletion: similarly, instrument PyXObject destructor
- Currently added 20 hooks to Python for these types:
  - Lists, Dictionaries, Strings, Integers, Long Integers

What's a Wedge?



Adding a Wedge

```
static DictDestRoutine dict_dtor_old = 0;

static void dict_dtor_wedge (PyObject* o)
{
  unsigned size;
  if (!dict_dtor_old) {
    fprintf(stderr, "dict_dtor_wedge: Invalid\n”);
    return;
  }

  size = o ? PyDict_Size(o) : 0;
  scoreEvent_(SCORE_DICT_DEALLOC, size, o);
  (*dict_dtor_old)(o);
}

/* Call from Python to set-up */
static PyObject*
catch_dict_dtors (PyObject* self, PyObject* args)
{
  char * s;
  if (!PyArg_ParseTuple(args, "s", &s)) return NULL;

  if (!dict_dtor_old) {
    dict_dtor_old = PyDict_Type.tp_dealloc;
    PyDict_Type.tp_dealloc = dict_dtor_wedge;
  }

  Py_INCREF(Py_None); return Py_None;
}
```

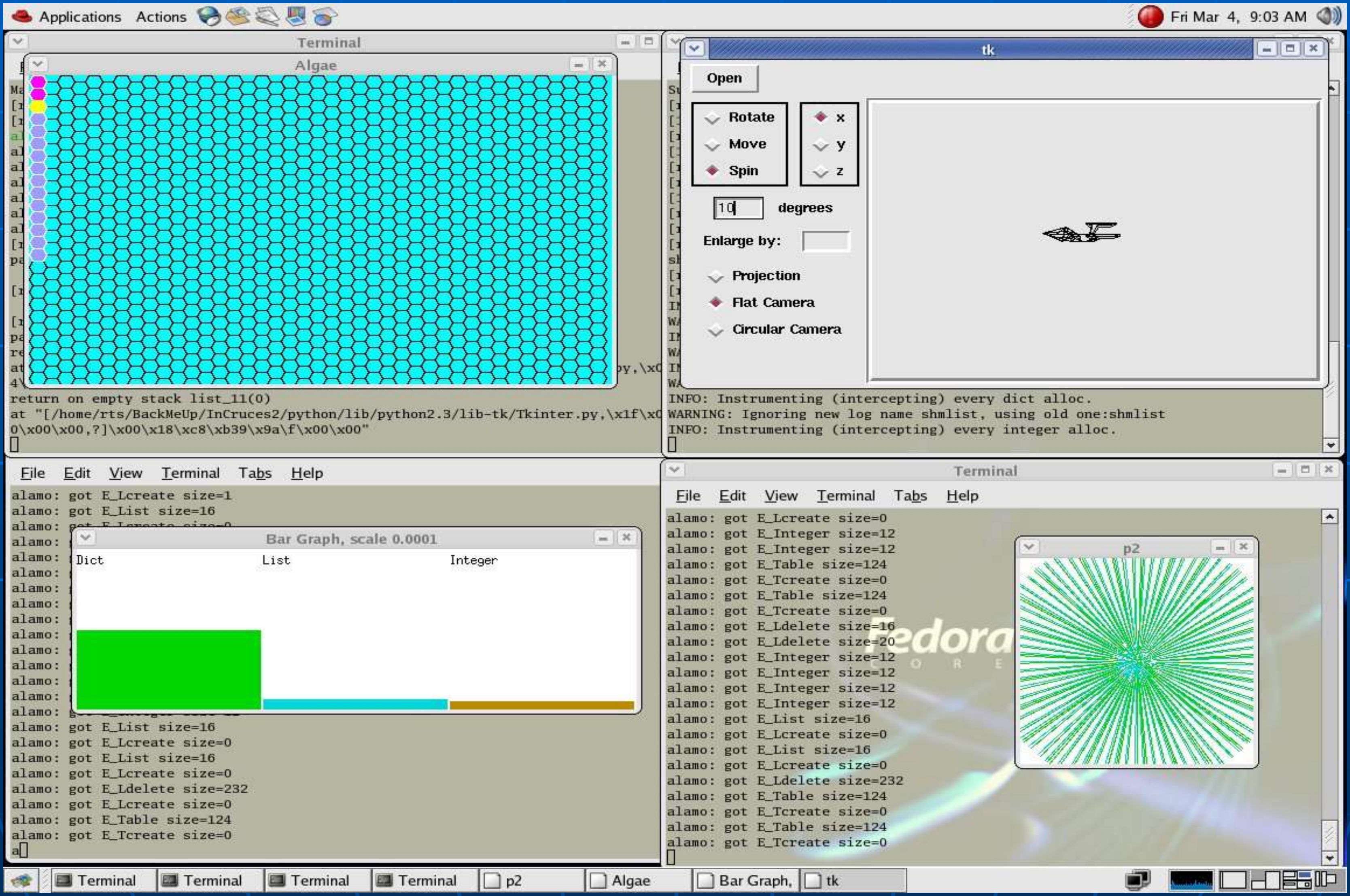
Multilingual Environment

- Python and Unicon in different processes
  - Python generates heavyweight events (Why? vs. top's lightweight)
  - Puts in an ULMA shared memory queue (double buffered)
  - Unicon reads event, and displays information in some visualization
    - Discussion: Can Python and Unicon exist in same process? How do they share information?

Example: Different Monitors

- *algae* shows call stack (perfect for generators)
  - Uses hexagons to approximate tree structure
- *nova* shows list construction events as a “circular clock”
  - Clock winds around as list construction events happen
- *barmem* shows construction events for lists, integers, dictionaries, strings, large integers
  - Gives idea how many objects you are constructing

Example: Python Sample Program with Unicon/Alamo Monitors



Open

○ Rotate
○ Move
● Spin

● x
○ y
○ z

10 degrees

Enlarge by:

○ Projection
● Flat Camera
○ Circular Camera

```
return on empty stack list_11(0)
at "[/home/rts/BackMeUp/InCruces2/python/lib/python2.3/lib-tk/Tkinter.py,\x1f\x0
0\x00\x00,?]\x00\x18\xc8\xb39\x9a\f\x00\x00"
```

```
INFO: Instrumenting (intercepting) every dict alloc.
WARNING: Ignoring new log name shmlist, using old one:shmlist
INFO: Instrumenting (intercepting) every integer alloc.
```

```
alamo: got E_Lcreate size=1
alamo: got E_List size=16
alamo: got E_List size=16
alamo: got E_Lcreate size=0
alamo: got E_List size=16
alamo: got E_Lcreate size=0
alamo: got E_Ldelete size=232
alamo: got E_Lcreate size=0
alamo: got E_Table size=124
alamo: got E_Tcreate size=0
a
```

```
alamo: got E_Lcreate size=0
alamo: got E_Integer size=12
alamo: got E_Integer size=12
alamo: got E_Table size=124
alamo: got E_Tcreate size=0
alamo: got E_Table size=124
alamo: got E_Tcreate size=0
alamo: got E_Ldelete size=16
alamo: got E_Ldelete size=20
alamo: got E_Integer size=12
alamo: got E_Integer size=12
alamo: got E_Integer size=12
alamo: got E_Integer size=12
alamo: got E_List size=16
alamo: got E_Lcreate size=0
alamo: got E_List size=16
alamo: got E_Lcreate size=0
alamo: got E_Ldelete size=232
alamo: got E_Table size=124
alamo: got E_Tcreate size=0
alamo: got E_Table size=124
alamo: got E_Tcreate size=0
```

Conclusion

- Wrote a real-time profiler for generating Linux “top”-like information
  - Need a few iterations to clean it up, but usable now
- Built a hybrid Python/Unicon system
  - Added Hooks to Python Virtual Machine that should potentially be put back in the main source tree
- Work still in progress, downloadable from http://www.rrc.com/downloads/PythonHooks

Future Work

- Move visualizations into Python
  - More events? More access to program state?
- Add support for threads
  - Don't currently support multi-threaded Python programs
- 3D Visualizations
  - Professor Jeffery currently working on collaborative virtual environment NSF grant, hoping we can reuse work
- Beowulf Cluster Monitoring: Can we scale?