University of Arizona, Department of Computer Science

CSc 620 — Assignment 6 — Due Fri, Dec 9, 14:00 — 30%

Christian Collberg
October 14, 2005

1 Introduction

Pick one of the projects below, pick a team to work on the project, and flesh out a design document (2-3 pages). The design document should specify the parts of the system that need to be built, how it will fit in with the rest of the system, how it will be evaluated, and a timeline for when different parts should be finished.

Here are important milestones for the rest of the semester:

Mon, Oct 17: Teams and projects should have been chosen. Open discussion in class about what needs to be done.

Wed, Oct 19: Design document due. Each group gives a 10 minute presentation of their project. Open discussion about interface issues between different projects.

Mon, Oct 24: First revision of project interfaces are due. Class discussion.

Wed, Oct 26: No class. Each team sees Christian for 10 minutes to discuss progress.

Wed, Nov 9: Each team gives a 10 minute class presentation about their progress. Class discussion.

Wed, Nov 23: Each team gives a 15 minute class presentation about their progress including a working demo. Class discussion.

Mon, Dec 5: Each team gives a 15 minute class presentation about their progress including a working demo. The first version of the project documentation is handed in and distributed to the class. Class discussion.

Fri, Dec 9, 14:00: Final presentation. All code and documentation is due.

These dates are subject to change. From now on, we will only have class as needed.

You should re-read the NSF proposal I handed out at the beginning of the semester, for more details on what we're planning to do.

Below is a list of the four sub-groups that I have envisioned for the project. The number of people in each group is just an estimate — obviously I want people to get involved in projects they have an interest in. No sub-group should probably have more than three people, though.

Every project should

1. have a working implementation;

2. have extensive documentation;
3. be thoroughly evaluated;
4. be integrated with the other projects.

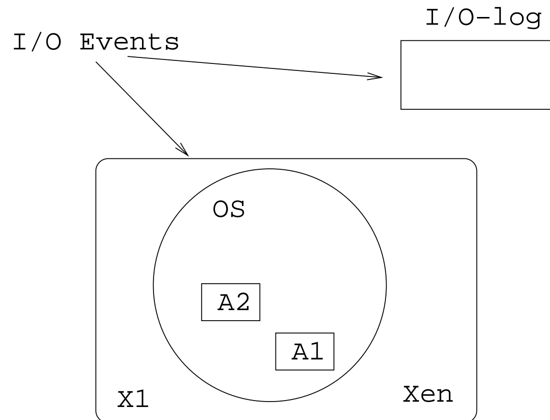
If you have your own idea about what you would like to do, I, of course, want to hear about it.

2 Virtual Machine Group [2 people]

Our basic architecture will be as follows.

- We will run inside a virtual machine (VM) environment.
- We will use the Xen VM, which can be downloaded from here: `http://www.cl.cam.ac.uk/Research/SRG/netos/xen`.
- We will "fork off" copies of the VM to support extensive profiling and reverse debugging.

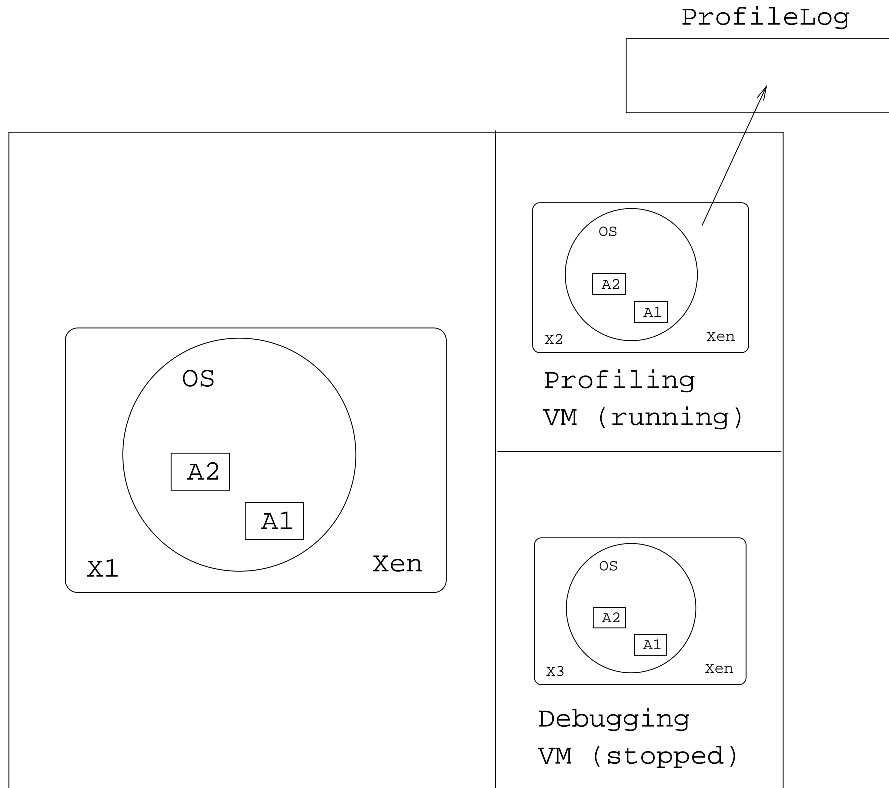
    1. For example, we might start off with the following situation at time T_0:

    The OS (Linux) is running inside Xen, and two application processes `A1` and `A2` are running within the OS.

    An I/O log keeps all external events for later replay.

    2. At time T_1 we create two copies of `X1`:

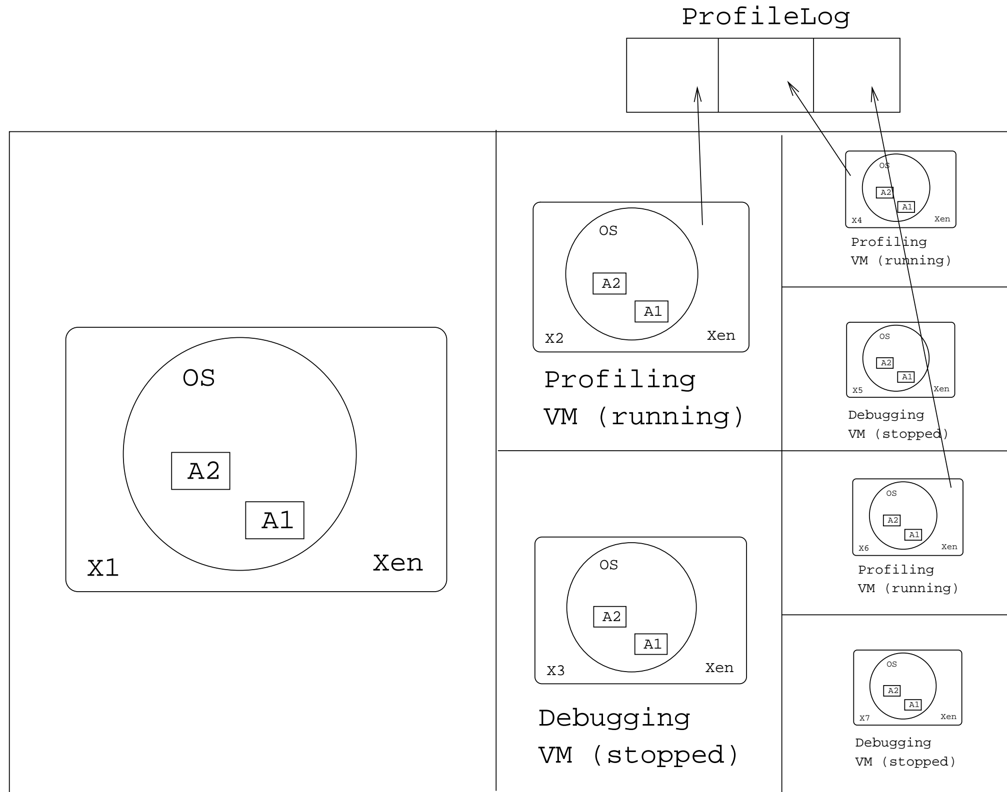


The idea is that `X1` is the process with which the user interacts. It runs at near full speed, and collects very little data. What little information it collects it displays using simple counters and dials, to give the user a rough idea of what's going on.

`X2` is a copy of `X1` that is used for profiling. It runs in a background, getting all the same I/O events as `X1`, but runs much slower, because it collects a bunch of profiling data. If the user detects a performance anomaly (from her interaction with `X1`) she can go back in time by switching to `X2`, which, at that time, has already gathered much of the necessary performance data.

`X3` is a copy of `X1` that is used for debugging. Just like when in Booth's system he forks off a process that just sits there until needed, `X3` just sits idle until we decide to go back in time.

3. At some future time T_3 we have created additional Xen instances `X4`...`X7`:



`X4` and `X6` are profiling VMs and run in the background, gathering performance data for some time period. If, for example, we create a new profiling VM every 10 seconds, `X2` would collect from time 0 to 10, `X4` from 10 to 20, and `X6` from 30-40. At the end of their time period, they are killed off.

`X5` and `X7` are idle debugging VMs, ready to be woken up whenever the user needs to go back in time. Like in Booth's system, they will have to be occasionally thinned out.

The virtual machine group is responsible for getting Xen up and running, and for maintaining the multiple Xen VMs that will be running.

3 Reverse Execution Group (3 people)

This group uses the VMs to implement the reverse execution environment.

As described above, the idea is to run the OS and application programs under Xen, and fork these off at various points during the execution, allowing us to return to a previous state.

The main point of this study is to find out if this approach is feasible on a multiprocessor. How many VMs can we fork off? How many processors do we need? What do we need to store to return to previous state, and then run forward to the point we're interested in (user I/O, network events,...)? Can we run backwards when several different application programs (in Java, C, Perl...) are running simultaneously? The latter would be cool, and novel!

The hope here is that using multiple Xen VMs will make reverse execution much easier to implement. Your task is to find out if this is true.

4 Profiling Group (3 people)

The main problem to solve here is to optimize the event handling to get as much data in (from event producers) and out (to disk and/or the visualization), as fast as possible. Any tricks are allowed, including the use of multiple disks, raw disk accesses, multiple processors, large amounts of shared/distributed memory, data compression, etc. The evaluation should focus on this performance, and with comparisons to previous systems.

Collect profiling data at all levels — from Xen itself and from CPU counters, JVMPI, instrumented binaries, etc. JVMPI may be too slow (adding an empty JNI call to every method in a program slows it down by a factor of 10) — instrumenting the bytecode or modifying the JVM might be a better strategy.

We are particularly interested in the use of dual core processors. Would it be possible for one core to generate event into the level 3 cache, such that the second core can process them, without having to go through memory?

5 Visualization Group (3 people)

Investigate one of the free game engines (Ogre seems like a good choice) to do visualization. Start with some very simple profiling visualizations, and then progress to some of the more advanced ones, below.

5.1 Heap Profiling and Visualization

Extract information from a heap (Java and C) and visualize its dynamic behavior. Allow common (and uninteresting) events and objects to be filtered out. Your documentation should contain an overview of previous heap visualization attempts.

You might be able to use the JVMPI/JVMTI system, or you may have to hack the Java runtime system to get out the information that you need.

I'm thinking that there are lots of interesting ways to filter the events, presenting different types of heap graphs in different ways. For example, the heap will contain may simple, small, transient objects, such as `java.lang.String`. It will contain a few long-lasting objects, such as hash tables. Again, they are often simple structures, such as hash tables of `java.lang.Integer` objects. And, finally, the heap may contain some long-lived complex linked structures, such as trees and graphs. The different types may be visualized in different ways. Transient objects probably shouldn't be drawn, just listed in a table or visualized in a bar-graph. Hash tables and other built-in Java classes could be visualized using some standard drawings. General graph structures, finally, could use standard graph drawing techniques (such as spring embedders). These techniques would, hopefully, allow huge heaps to be visualized effectively.

You may be able to use Stephen Kobourov's graph drawing system for visualizing changing graphs.

5.2 Trace Profiling and Visualization

Extract trace information from a Java and/or C program and visualize it (dynamically, as the program is running). Allow the trace to be viewed at different levels of detail — i.e. it should be possible to get an "overview trace" showing, for example, which modules/classes/packages the program spends its time in, or a more detailed trace showing the call graph as it is explored.

Allow common (and uninteresting) parts of a trace to be filtered out. Your documentation should contain an overview of previous trace visualization attempts.

The main part of this study is to examine current visualization tools (Blender, VRML, Ogre, etc.) to see which will provide the necessary abstractions and performance.

5.3 Static Visualization

Use SandMark to extract the static structure of Java programs, or use PLTO (or similar tool) for C. Visualize it using one of the geographic metaphors we've discussed in class.

The main part of this study is to examine current visualization tools (Blender, VRML, Ogre, etc.) to see which will provide the necessary abstractions and performance.

6 Submission and Assessment

1. You will give a final presentation of the project on Fri, Dec 9, 14:00.
2. We will also have 1-3 intermediate presentations to the class.
3. You should hand in all source code and documentation by Fri, Dec 9, 14:00.
4. The code should be working, neatly formatted, commented, and documented.
5. The project should build cleanly, simply by typing "make".
6. Each group should hand in a separate document (roughly 10 pages) describing the project, its goals, previous work, the project implementation, and evaluation results. The document must be written in LaTeX.
7. The project should have a web page containing a description, code to download, screen shots (if appropriate), etc.

All files (source code, documentation, web pages, etc.) should be maintained under the Peacock CVS tree. Every subdirectory (regardless of whether it contains source code, documentation, web pages, etc.) should have a README file describing its purpose and contents and a Makefile building and installing the artifact.