Tools: A Short Story

Computer Science 252

1 pixie, pixstats, prof

Documentation: man pages, pixie_manual.ps

Where to Look: cs252/f94/sim/Pixie/{pixie_manual.ps, csim.c, redir3.sh}

Pixie is a program that traces the execution of a program. No recompilation is necessary; pixie takes a binary as input and produces an annotated binary as output.

To generate a pixie file, type:

`pixie prog`

where `prog` is your executable.

This generates two files: `prog.Addrs` and `prog.pixie.` The `prog.Addrs` file contains the addresses of all of the basic blocks from the executable; basically, a set of (block number, offset) pairs. `prog.pixie` is the new executable. It now contains extra instructions to record how often each basic block is executed.

To generate dynamic instruction frequency, etc., you must now run the annotated binary, `prog.pixie`.

(Note: if it used to take two inputs, `prog -f file -o outfile`, run it like this: `prog.pixie -f file -o outfile`.)

This will create one last pixie file, called `prog.Counts.` This contains information on how many times each basic block was entered, and therefore can be used to determine dynamic program behavior.

Two additional tools, called `pixstats` and `prof`, exist to help analyze program behavior. Example output from `pixstats`:

```
4505 (1.000) cycles (0.00018s @ 25.0MHz)
4505 (1.000) instructions
 548 (0.122) basic blocks
  31 (0.007) calls
 972 (0.216) loads
 549 (0.122) stores
1521 (0.338) loads+stores
1521 (0.338) data bus use
  53 (0.012) partial word references
 448 (0.099) branches
1037 (0.230) nops
```

There is a lot more output by `pixtstats`, which you should feel free to look at sometime. One particularly useful flag is `-disassemble`. Try it out and see what happens.

The other tool, `prof`, also gives execution profiling information, and also tells you where you are spending your time in your C code. This can be a useful tool to most anyone (well, most any computer scientist). I'll leave it to the reader to try out `prof`.

Now, let's say we want to generate an instruction and data trace for a program (again, program `prog`). Run pixie again, but this time, as follows:

```
pixie prog -idtrace -idtrace_file 3
```

The `-idtrace` flag tells pixie to generate traces for both the instruction and data streams, and the `-idtrace_file 3` option redirects output to file descriptor 3.

Run the pixified program again.

```
prog.pixie
```

Now, use the script 'redir3.sh' to generate a trace file.

```
redir3.sh prog.pixie
```

This script redirects the trace to a file called `prog.pixie.trace.` This is a binary file which gives you all the necessary information for tracing. How do we get at that file?

Well, luckily we have a little skeleton of a program called `csim.c`. It reads in the trace file and interprets it. This could be used as the beginnings of a cache simulator, or simply as something to generate more 'human-readable' trace files. Here is an example:

```
BASIC_BLOCK:    400140  0
WORD_READ:      ff88d0  3
WORD_WRITE:     1298    2
WORD_WRITE:     129c    2
WORD_WRITE:     1290    0
WORD_WRITE:     ff88cc  1
WORD_WRITE:     1294    0
BASIC_BLOCK:    4001b0  1
WORD_WRITE:     ff78a4  0
WORD_WRITE:     ff88b8  0
```

This was generate from the following code trace (produced by `pixstats`, with the `-disassemble` flag on).

```
0x00400140      0x8fa40000      lw      a0,0(sp)
0x00400144      0x3c1c1001      lui     gp,0x1001
0x00400148      0x279c9130      addiu   gp,gp,-28368
0x0040014c      0x27a50004      addiu   a1,sp,4
0x00400150      0xaf858168      sw      a1,-32408(gp)
0x00400154      0x24a60004      addiu   a2,a1,4
0x00400158      0x00041080      sll     v0,a0,2
0x0040015c      0xaf80816c      sw      zero,-32404(gp)
0x00400160      0x00c23021      addu    a2,a2,v0
0x00400164      0x27bdffe8      addiu   sp,sp,-24
0x00400168      0xaf868160      sw      a2,-32416(gp)
0x0040016c      0xafa00014      sw      zero,20(sp)
0x00400170      0x0c10006c      jal     0x4001b0
0x00400174      0xaf848164      sw      a0,-32412(gp)
```

2 dinero

Documentation: man pages (found in tar file)

Where to Look: cs252/f94/sim/dinero/{dineroIII.3.4.tar}

The program `dinero` is a cache simulator written by Mark Hill, a former U.C. Berkeley graduate student, now teaching at Wisconsin. It is fairly straightforward to use, as shown in this example:

```
\% dineroIII -b32 -d8192 -i8K -a1 < input.file

CMDLINE: dineroIII -b32 -d8192 -i8K -a1
CACHE (bytes): blocksize=32, sub-blocksize=0, Usize=0, Dsize=8192, Isize=8192.
POLICIES: assoc=1-way, replacement=l, fetch=d(1,0), write=c, allocate=w.
CTRL: debug=0, output=0, skipcount=0, maxcount=10000000, Q=0.

---Simulation begins.
---Simulation complete.

 Metrics                 Access Type:
 (totals,fraction)       Total   Instrn   Data    Read   Write    Misc
 -----------------       ------  ------  ------  ------  ------  ------
 Demand Fetches            1000     470     530     234     296       0
                         1.0000  0.4700  0.5300  0.2340  0.2960  0.0000

 Demand Misses               59      26      33      16      17       0
                         0.0590  0.0553  0.0623  0.0684  0.0574  0.0000

 Words From Memory          472
 ( / Demand Fetches)     0.4720
 Words Copied-Back          136
 ( / Demand Writes)      0.4595
 Total Traffic (words)      608
 ( / Demand Fetches)     0.6080
```

Certainly, `dinero` has a lot of features. Some of the parameters that can be varied include total size of cache, block size, associativity, replacement policy, write strategy, etc. I encourage the interested reader to pursue this further in the man page found in the .tar files.

One problem with `dinero` is that it was designed for only one level of the memory hierarchy. In other words, it cannot arbitrarily simulate n levels of cache. Fortunately, though, `dinero` does have a way of simulating a secondlevel cache. The script `two-level.awk` takes the output from the first run (which simulates a firstlevel cache), and uses the miss information to perform a second run for the secondlevel cache. If you're interested, feel free to play around with the script.

One item that may be of interest is how to generate the input trace that `dinero` requires for simulation. The `dinero` input format is as follows:

```
LABEL = 0       read data
        1       write data
        2       instruction fetch
        3       escape record (treated as unknown access type)
```

```
4       escape record (causes cache flush)
```

Therefore, each line of the ascii input file is of the form (LABEL, address), where label is from 0 to 4 (as above), and address is where the read, write, or fetch occurs.

Here's an example input file:

```
2 00400140
0 8fa40000
2 00400144
2 00400148
2 0040014c
2 00400150
1 af858168
2 00400154
2 00400158
2 0040015c
1 af80816c
2 00400160
2 00400164
2 00400168
1 af868160
2 0040016c
1 afa00014
2 00400170
2 00400174
1 af848164
```

Note that this input file should correspond to the `pixie` example output from earlier in the handout.

Converting a `pixie` trace to this format, with the help of the `csim.c` code mentioned earlier in the handout, should be a relatively straightforward task.

3 qpt

Documentation: README, man pages (in .tar file)

Where to Look: cs252/f94/sim/qpt/{qpt}

`qpt` is another program written by a former Berkeley student (Jim Larus), now teaching at Wisconsin. In essence, it is very similar in funtionality to `pixie`, and therefore I won't be spending much time talking about it.

One difference that might be important to people is that `qpt` runs on *both* Sparcs and Mips machines, whereas `pixie` is a DECMipsonly tool.

4 CPROF

Documentation: cprof.man.ps, xcprof.man.ps xcprof.users.guide.ps, all in .tar file (under doc/); also, spec92.ps, a paper about a study that uses CPROF to analyze and improve the cache behavior of the SPEC 92 benchmarks.

Where to Look: cs252/f94/sim/cprof/{CPROF.tar}

Not surprisingly, `CPROF` is a tool produced by a former Berkeley student who is now (yes!) working at Wisconsin (this time, it's David Wood). At least Berkeley students *used* to be productive...

What does `CPROF` do? Well, it is a program that analyzes cache behavior. That sure sounds a lot like `dinero`, doesn't it? Sure, but `CPROF` provides you with more information on the types of misses the program incurs. Misses are broken down into the following categories: compulsory, capacity, and conflict.

This program gives you information at the source level. In other words, it will tell you that one particular part of your C code is causing a hell of a lot of capacity misses, so you might think about moving addresses around so as to avoid hitting the same line in the cache.

A short example can be found in the demo directory, and is probably a good place to start. There is also a nice graphical interface known as `xcprof`.

5 spim

Documentation: spim.ps, in spim directory (listed below); also in .tar file

Where to Look: cs252/f94/sim/spim/{spim.tar.gz}

`Spim` is an R2000 simulator, written by Jim Larus, the author of `qpt`. It simulates the full machine, including registers, pipeline, etc. Simply give it an executable or assembly file, and it will run it to completion. By stepping through the program, one can see the behavior of the program in terms of how it updates machine state.

In a sense, `spim` looks a lot like a debugger. It loads in your executable (or assembly file), lets you set break points, step through code, etc. This is a really useful tool if you want to learn about one of the early MIPS architectures. Also, if you are interested in modifying a machine's organization to see what the impact on performance is, this is a good place to start.

A "nice" graphical interface called `xspim` exists and can be used instead of the textual `spim` for viewing progress.

A cycle-level simulator known as `cl-spim` is also available, which further breaks down machine behavior, accounting for every cycle that the machine uses, including bus transactions, etc. If there is an interest in this, tell me and I can make it available.

6 Cacti

Documentation: A paper on the model, wrl93.5.ps

Where to Look: cs252/f94/sim/cacti/{cacti.tar, wrl93.5.ps}

A tool that takes input parameters such as (cache size, line size, and associativity) and produces the following output assuming a 0.8 micron process:

```
% cacti 16384 64 1
```

```
Cache Parameters:
  Size in bytes: 16384
  Number of sets: 256
  Associativity: 1
  Block Size (bytes): 64
Access Time: 5.49839e-09
Cycle Time:  7.62046e-09

Time Components:
[a bunch deleted]
```

I'm not sure if this will be useful, but hell, it's a tool.

7 shade, spixtools

Documentation: `shade.v8/doc/intro/intro.ps`, `shade.v8/man/man.body.ps`, `spixtools.v8/doc/intro/intro.ps`

Where to Look: You must get an account from me on the now cluster to gain access to shade and corresponding documentation.

Last, but certainly not least, is a program-tracing utility out of Sun Labs known as `shade`. You could consider this Sun's version of `pixie` (but that would be greatly underestimating it's power and flexibility).

(There is so much to `shade` and the accompanying `spixtools` that I could have spent the whole hour discussing them. Instead, I am going to leave that to the reader. Why? I'm not lazy. Well, maybe I am. Moreover, I sure am tired. Anyway...)

A quick summary of `shade`:

- dynamically translates application code into host machine code (plus annotations), then executes. New code is cached to reduce overhead.
- analyzer must be written by user (that's you), to collect information on whichever instructions the user (you again) wishes.

The next page contains a simple example of how this might work.

A quick summary of `spixtools`: it's a collection of programs which generates different information on instruction-level profiles.

- `spix`: instruments program (ala `pixie`) for basic block counts
- `spixstats`: a lot like `pixstats`
- `sdas`: disassembles app and annotates assembly (ala `pixstats -disassemble`)
- `sprint`: prints source of app, annotates it with instruction counts (more like `prof`).
- `sadd`: adds or subtracts `spixcounts` files.
- `snap`: allows checkpointing.
- `score`: creates `spixstats` file from core dump.

Other niceties included within the `shade` packagem, with source available:

- `icount`: simple example of a shade program, counts instructions
- `cachesim`: a cache simulator (yes!)
- `window`: all sorts of info on register window usage
- the list goes on...