Arquitectura de Sistemas

Análisis de rendimiento

Gustavo Romero López

Updated: 4 de abril de 2019

Arquitectura y Tecnología de Computadores

Objetivos

- Aprender los principios de la optimización de código:
  1. Escoger el mejor algoritmo.
  2. Utilizar la mejor implementación.
  3. Optimizar.
- Instrumentación: código analizándose sí mismo.
  - Propensa a errores.
  - ¿Mide lo que en realidad creemos?
  - Intenta evitarla.
- Software de análisis de rendimiento:
  - Instrumentación: `gprof`
  - Simuladores: `valgrind`
  - Muestreo (estadísticos): `oprofile` y `perf`

gprof

- Híbrido entre instrumentación y muestreo.
- Forma de uso:
  1. Compilar con la opción "`-pg`".
  2. Al ejecutar se crea un fichero llamado `gmon.out`.
  3. Analizar los resultados de `gmon.out` con `gprof`.
- Tutorial:
  http://www.thegeekstuff.com/2012/08/gprof-tutorial/

`valgrind`: http://valgrind.org/

- Máquina virtual que emplea técnicas de interpretación dinámica (JIT).
- Colección de herramientas para detección de errores y análisis de rendimiento.
- Inconvenientes:
  - Ejecución entre 5 y 20 veces más lenta.
  - Alta utilización de memoria.
- Ventajas:
  - No es necesario modificar los binarios.
  - Buena interfaz gráfica: `kcachegrind`/`qcachegrind`.
- Manual: http://valgrind.org/docs/manual/manual.html

`oprofile`: http://oprofile.sourceforge.net/news/

- Analizador de rendimiento estadístico basado contadores hardware y eventos.
- Uso habitual:
  1. Recopilar datos con "`operf`", "`ocount`" o "`opgprof`".
  2. Analizar los resultados con "`opreport`" o "`opannotate`".

perf: https://perf.wiki.kernel.org/index.php/Main_Page

- Analizador de rendimiento estadístico basado contadores hardware y eventos.
- Capaz de registrar gran número de eventos: "`perf list`".
- Uso habitual:
  1. Recopilar datos con "`perf record -e 'evento' -- ./exe`".
  2. Analizar los resultados con "`perf report`".
- Manuales:
  - `https://perf.wiki.kernel.org/index.php/Tutorial`
  - `http://sandsoftwaresound.net/perf/perf-tutorial-hot-spots/`

Primer ejemplo: `facil.cc`

facil.cc

```
const int N = 1 << 25;

int f1()
{
  int r = 0;
  for (int i = 0; i < N; ++i)
    r += i;
  return r;
}

int f2()
{
  int r = 0;
  for (int i = 0; i < 2 * N; ++i)
    r += i;
  return r;
}

int main()
{
  return f1() + f2();
}
```

- Pruebe a analizar su rendimiento con...
  1. `gprof`
  2. `valgrind`
  3. `perf`
- Este es un programa de prueba que no tiene ningún misterio pero le ayudará a empezar a prácticar con los analizadores de rendimiento.

Memoria y caché: multiplicación de matrices: `ijk.cc`

`ijk.cc`

```
void f1()
{
  for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
      a[i][j] = 0;

  for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
      for (int k = 0; k < N; ++k)
        a[i][j] += b[i][k] * c[k][j];
```

- ¿Cuál de las formas de combinar los índices i, j y k nos permite multiplicar matrices de manera más rápida?
- ¿Qué analizador de rendimiento es más apropiado utilizar en este caso: `valgrind` o `perf`?

Algoritmos recursivos: `fib.cc` y `ackermann.cc`

- ¿Se pueden implementar de manera iterativa? ¿Merece la pena?
- Compare las versiones 2 y 3 de Fibonacci... ¿Cuál es mejor y por qué?
- ¿Qué imlementación de la función de Ackermann es mejor y por qué?

Algoritmos iterativos: `matrix.h` y `matrix.cc`

- Analice este programa de cálculo con matrices e intente mejorarlo.

Código automodificable: `selfmod.cc`

- Estudie si es rentable o no el cambio que este programa hace en sí mismo para logra una mayor eficiencia.

Ejercicios de exámenes: `axb.cc` y `struct.cc`

- `axb.cc` y `struct.cc` son ejercicios que se ha pedido resolver en exámenes.
- Intente escribir mejores versiones.
- GCC dispone de una opción `-fopt-info` que nos aporta información interna del compilador sobre las optimizaciones aplicadas a nuestro código fuente.
- Una vez concluido su intento, eche un vistazo a algunas de las soluciones aportadas por profesores y alumnos: `axb-etal.cc` y `struct-etal.cc`