Trabajo Practico 1: Conjunto de Instrucciones MIPS

Guido Laghi, *P. 82.449*
guido321@gmail.com

Sebastian L. Perez, *P. 84.379*
sebastian.leo.perez@gmail.com

Sergio Matias Piano, *P. 85.191*
smpiano@gmail.com

1er. Cuatrimestre de 2013
66.20 Organizacion de Computadoras
Facultad de Ingenieria, Universidad de Buenos Aires

Resumen

Este informe sumariza el desarrollo del trabajo practico 1 de la materia Organizacion de Computadoras (66.20) dictada en el primer cuatrimestre de 2013 en la Facultad de Ingenieria de la Universidad de Buenos Aires. El mismo consiste en la construccion de un sistema minimalista de ordenamiento de archivos y el analisis de performance y perfilado del mismo.

Índice

Parte I

Desarrollo

1. Introduccion

El proceso de compilacion de un programa codificado en C consiste en un pipeline que genera finalmente codigo objeto para una arquitectura objetivo. Este pipeline analiza y traduce el codigo a diferentes representaciones de los algoritmos contenidos en el, de manera de terminar generando codigo objeto que finalmente se linkea en un ejecutable valido para la arquitectura.

Debido a que el lenguaje C es un lenguaje de alto nivel, varios de los conceptos propios de este lenguaje deben ser mapeados a diferentes estrategias que pueden ser ejecutadas en el codigo objeto, que no es mas que el conjunto de instrucciones que la arquitectura objetivo provee. Estos mapeos muchas veces pueden implementarse de diferentes maneras, cada una con sus ventajas y desventajas, y si bien el compilador realiza una muy buena tarea en determinar la mejor manera en la que traducir estos conceptos en instrucciones de bajo nivel, los algoritmos utilizados no garantizan que la traduccion sea de manera de generar el codigo objeto mas eficiente y compacto que se podria tener.

La alternativa que se propone explorar en el presente trabajo es la implementacion directa en assembly de las rutinas del sistema que se esta construyendo. Esta alternativa tiene una caracteristica fundamental: al tener acceso al conjunto de instrucciones que ejecuta la arquitectura potencialmente se puede desarrollar una solucion optima, minimizando el acceso a memoria, la cantidad de instrucciones a ejecutar, etc.

Por otro lado, la implementacion en assembly de un sistema no trivial es una tarea complicada. La dificultad principal radica en que es tarea del programador implementar y controlar muchos de los mecanismos de bajo nivel que constituyen el *como* de la solucion, en vez de concentrarse en el *que*. Algunos ejemplos son el manejo del stack, la implementacion de convenciones de llamadas de subrutinas, el uso de registros compartidos, entre otros.

2. Implementacion

El sistema que resuelve el problema planteado en el enunciado (disponible en el anexo A) fue implementado mayoritariamente en C, y parte en lenguaje MIPS como lo indicaba el enunciado. La diferencia fundamental radica en la implementacion de las rutinas de ordenamiento a traves del algoritmo *shellsort* directamente en assembly. El codigo fuente de la solucion esta

disponible en el anexo C; por cuestiones de espacio y prolijidad solo se incluyen los archivos relacionados con la implementacion en assembly de la solucion desarrollada.

Cabe destacar que la solucion en assembly definitivamente no es la solucion optima. Existen algunas mejoras posibles para disminuir la cantidad de instrucciones a ejecutar o los accesos a memoria: por ejemplo, se podrian reordenar las instrucciones de algunos de los bloques de los procedimientos implementados para eliminar algunas de las instrucciones de salto incondicional, o se podrian utilizar de mejor manera los registros salvados (s0-s8) para eliminar la necesidad de restaurar desde el stack algunos de los registros en uso (particularmente a0, a1 y a2 en la implementacion de $shell\_sort\_s$). Sin embargo, dado que la solucion que se presenta contiene el 40 % de las instrucciones del equivalente implementado en C compilado sin optimizaciones, a los efectos del trabajo practico consideramos suficientemente optima la implementacion propuesta.

A continuacion enumeraremos algunas consideraciones de diseño tomadas al implementar cada una de las funciones involucradas en el algoritmo de ordenamiento.

2.1. shell_sort_s

Esta funcion implementa el ordenamiento de una tabla de strings a traves de *shellsort*. El metodo de ordenamiento fue implementado en su forma mas sencilla a traves de un algoritmo recursivo. El stack frame tipico de la funcion se describe en la imagen adjunta al final del informe.

Se reservan 32 bytes para el area de SRA: 4 bytes para salvar el registro ra en llamadas a otras funciones (ya que no es leaf), 4 bytes para salvar el registro gp, 4 bytes para preservar el registro fp y finalmente 20 bytes para el uso interno de los registros S0, S1, S2, S3, S4 que se utilizan para calculos de la funcion.

Por otro lado, dado que las funciones invocadas desde esta no poseen nunca mas de dos argumentos, se reservan $4x4$ bytes en concepto de ABA (respetando la ABI) para ser

utilizados por las funciones llamadas.

Esta funcion invoca a strcasecmp como a data_swaper.

3. Compilacion

Se instrumento un *makefile* para ejecutar las instrucciones adecuadas de compilacion para los distintos escenarios requeridos. La tarea *make* compila

individualmente cada uno de los archivos fuente de extension *c* y *S* a traves del ejecutable *gcc*. Los comandos utilizados para compilar cada uno de estos archivos fuente son los siguientes:

```
gcc -c -o build/obj/buffer.o source/buffer.c -I./source -Wall
gcc -c -o build/obj/data.o source/data.c -I./source -Wall
gcc -c -o build/obj/clargs.o source/clargs.c -I./source -Wall
gcc -c -o build/obj/cltext.o source/cltext.c -I./source -Wall
gcc -c -o build/obj/tp1.o source/tp1.c -I./source -Wall
gcc -c -o build/obj/bubblesort.o source/bubblesort.c -I./source -Wall
gcc -c -o build/obj/shellsort.o source/shellsort.c -I./source -Wall
```

De este listado, cada una de las invocaciones obedece a la siguiente estructura de argumentos:

-c Compila o ensambla el codigo fuente pero no corre el linker. Por lo tanto la salida corresponde a un archivo objeto por cada archivo fuente.

-o Especifica cual sera el archivo de salida sea este un archivo objeto, ejecutable, ensamblado o codigo preprocesado de C.

-Wall Activa todos los mensajes de warning.

-I Agrega el directorio especificado a la lista de directorios buscados para los archivos header

El resultado de la ejecucion de estos comandos es que se generan archivos objeto para cada fuente, listos para ser linkeados, en el directorio *build/obj*. Para realizar este ultimo paso, se invoca nuevamente a *gcc* con un ultimo comando:

```
gcc -o build/tp1 build/obj/buffer.o build/obj/data.o build/obj/clargs.o b
```

El comando linkea todos los archivos objeto en un ejecutable final, *build/tp1*.

4. Analisis de tiempos de ejecucion

En las siguientes secciones detallaremos el proceso de analisis de tiempo de ejecucion realizado para los diferentes casos contemplados.

4.1. Analisis previo

El analisis de tiempos de ejecucion se focalizara en comparar la performance de la implementacion en assembly descripta anteriormente contra diferentes versiones de esencialmente el mismo sistema.

Primero se comparara el desempeño de los algoritmos escritos en C. utilizando todas las optimizaciones disponibles por el compilador. Se espera nuevamente encontrar algunas mejoras, aunque pequeñas, con respecto a lo que puede entregar el compilador.

Luego se comparara el algoritmo escrito en C contra el mismo en lenguaje ensabmblador. Por un lado, al compilar esta solucion sin ninguna optimizacion del compilador se espera obtener ganancias de performance apreciables, debidas mayormente al mejor manejo de los accesos a memoria logrados por el ejecutable desarrollado en assembly.

4.2. Experimentos

Se realizaron pruebas utilizando los archivos provistos por la catedra sobre un entorno de ubuntu actual y sobre el emulador GXemul. Obtuvimos los siguientes resultados:

Archivo	Tiempo de ejecucion
Alice	13,107s
Beowulf	20,749s
Cyclopedia	3m27,744s
EL Quijote	60m32,782s

Cuadro 1: Tiempos en Ubuntu para BubbleSort

Archivo	Tiempo de ejecucion
Alice	0,422s
Beowulf	0,537s
Cyclopedia	1,920s
EL Quijote	5,675s

Cuadro 2: Tiempos en Ubuntu para ShellSort

Archivo	Tiempo de ejecucion
Alice	15m17,180s
Beowulf	27m11,102s
Cyclopedia	135m
EL Quijote	*timeelapsed*

Cuadro 3: Tiempos en GXemul para BubbleSort

Archivo	Tiempo de ejecucion
Alice	7,281s
Beowulf	9,500s
Cyclopedia	29,234s
EL Quijote	2m8,613s

Cuadro 4: Tiempos en GXemul para BubbleSort

Se observaron todos los resultados esperados. Se noto una mejoria en los tiempos del codigo assembly respecto al codigo C. El analisis grafico de los datos se puede apreciar adjunto al final de este informe.

5. Conclusiones

La implementacion en assembly del trabajo practico, a pesar de que los algoritmos son relativamente simples y sencillos de desarrollar, fue relativamente complicado. El entorno de desarrollo en la plataforma MIPS agrego complejidad a la hora de implementar, administrar y controlar la infraestructura de los llamados a funciones y otros artefactos basicos de integracion. Por otro lado, el codigo de ensamblador no provee muchas de las abstracciones basicas que se dan por sentado en los lenguajes de alto nivel, como bloques condicionales y conversion de tipos, lo que hizo que la implementacion del algoritmo sea aun mas dificultoso. La ausencia de mecanismos de seguridad presentes en lenguajes compilados, como la comprobacion de tipos, prototipos funcionales y declarativas en la asignacion de nombres a variables incremento la cantidad de bugs con los que desarrollamos, lo que dio a lugar a mas trabajo de depuracion y testing.

Analizando el resto de los resultados concluimos que la version implementada en C con *shellsort*, incluso cuando no se aplicaron optimizaciones de compilador, era un orden de magnitud mas eficiente que la version implementada en assembly.

Como conclusion final podemos decir que si bien la implementacion de MIPS nos ofrece un mejor rendimiento de los algoritmos desarrollados, su dificil implementacion hace que debamos evaluar si deseamos priorizar el rendimiento a costo de tiempo de desarrollo. En caso de necesitar una mejora crucial de rendimiento la mejor solucion es la arquitectura MIPS, mientras que si deseamos un desarrollo mas sencillo (y por lo tanto mas breve) la mejor opcion es lenguaje en C.

Parte II

Apendice

A. Enunciado original

B. Codigo fuente