Hecho por: Bernardo Daniel Treviño Caballero

Diseño de la memoria de ejecución

Experimento: ¿Estructura con diccionarios versus clase de memoria?

Como un futuro candidato de graduación para ciencias computacionales, se elaboró este experimento para tomar la mejor decisión en cuanto a qué estructura de datos utilizar para la memoria ejecutable de la máquina virtual de un compilador.

Estructura de datos con diccionarios

Durante el diseño de la memoria de ejecución se contemplaba una estructura que funcionara como el monolito abstracto que refleja la memoria, sin serlo. Esto quiere decir que había que ingeniárselas para formar una estructura de datos rápida y eficiente para soportar lectura y escritura de datos, que a su vez pudieran ser muchos.

Pensando fuera de la caja se pensó en cómo sacarle provecho a las estructuras conocidas y muy usadas en Python (lenguaje de desarrollo de la máquina virtual). Estas estructuras son diccionarios, que se implementan como una tabla de hash y arreglos. Después de pensar en cómo funciona la memoria en ejecución, se diseñó una memoria virtual que consiste en un arreglo que contiene 4 elementos. Cada elemento corresponde a una de las secciones de la máquina virtual: global, local, temporales y constantes. La sección global y constante, consiste únicamente en un diccionario que contendrá todos los valores. Por otro lado, la sección local y temporal, se comporta muy interesante, puesto que hay llamadas a funciones de las cuales se tiene que crear un nuevo espacio en memoria. Con esta lógica, se optó entonces por una lista de diccionarios para estas secciones. El primer diccionario de la lista correspondería al *Main* del programa. Cuando se llama a una función, se creará otro diccionario que se añadirá a al final de la lista. De esta manera, el scope en memoria sería el equivalente a un diccionario.

Para que las secciones temporal y local funcione como se espera de una memoria de ejecución, se comparó su forma de trabajar con la de una pila. Así pues, cada vez que se crea un nuevo espacio en memoria (diccionario) se va insertando en el tope de la pila y sobre este scope (diccionario) se trabaja. Una vez que se termine de ejecutar el código de la función, se elimina el diccionario del tope de la lista, haciendo que se regrese o “despierte” la memoria con la que se estaba trabajando antes de la llamada.

En base a lo antes dicho, la estructura de la memoria es la siguiente:

memory = [{}, [{}], [{}], constants]

Donde constants es el diccionario de constantes obtenidos del archivo que contiene el código intermedio.

Clase Memoria

Otra opción para la memoria de ejecución es la que se ha recomendado mucho en la clase. Esta clase consiste en crear una estructura que en base a un cálculo aritmético de la memoria virtual se obtenga la posición en los arreglos en donde se debe almacenar el valor de la variable.

Esta estructura supone que contenga una serie de arreglos dependiendo del tipo de dato y alguna forma de control para saber qué espacios ya están ocupados. Además, se debe de cuidar de que no se desperdicien espacios del arreglo. Lo favorable de esta estructura es que es muy ordenada, no ocupa llaves y es muy parecido al monolito visto en clase, salvo que no tiene espacio que no se usa. Esta estructura sería muy favorable para lenguajes no dinámicos.

La siguiente implementación [1] fue obtenida por parte de Maribel Pastrana Brito y Norma Elizabeth Morales Cruz para ser usadas para fines de este experimento.

```
class Memoria:
    def __init__(self, cuadrRetorno, arrEntero, arrString, arrChar, arrDecimal, arrBool,
        arrEnteroTemp, arrStringTemp, arrCharTemp, arrDecimalTemp, arrBoolTemp):
        self.arrEntero = []
        self.arrDecimal = []
        self.arrString = []
        self.arrChar = []
        self.arrBool = []
        self.arrEnteroTemp = []
        self.arrStringTemp = []
        self.arrCharTemp = []
        self.arrDecimalTemp = []
        self.arrBoolTemp = []
        self.cuadrRetorno = 0
        self.listaMem = [arrEntero, arrString, arrChar, arrDecimal, arrBool, arrEnteroTemp,
            arrStringTemp, arrCharTemp, arrDecimalTemp, arrBoolTemp]
```

Problema: La estructura con diccionarios es ineficiente en tiempo comparado con la clase Memoria.

Para determinar si la hipótesis anterior es cierta se optó por correr 2 programas del lenguaje Arcadame en la máquina virtual. Una versión de la máquina virtual usará la implementación con diccionarios, mientras que la otra tendrá la clase memoria.

Para evaluar la rapidez se usará el timer de la biblioteca timeit de Python, considerado el más exacto. Además, se sabe que hay lecturas y escrituras en memoria, por lo tanto se medirán ambas operaciones separadamente.

Las pruebas se llevarán a cabo en dos máquinas virtuales diferentes que corren Arcadame. Ya se tenía la máquina virtual que contenía la estructura de diccionarios. Sin embargo, se tuvo que adaptar la versión de la máquina virtual para que pudiera soportar la clase Memoria obtenida por el otro equipo de la clase. También, la máquina virtual implementa dos funciones, uno para lectura y otro para escritura por la memoria. Estos métodos se vieron afectados para soportar ambos tipos de memoria.

Los métodos de acceso y asignación

El primero es el que se usa con la estructura de diccionarios. Lo que aparece comentado fue para hacer más real las pruebas, puesto que solo se usarán operadores aritméticos y lógicos y no hay datos dimensionados. Lo función de lo comentado es poder identificar los temporales indirectos (usados en datos dimensionados).

```
def assignValueInMemory(memoryKey, value):
    global memory
        ##if (isinstance(memoryKey, str)):
        ##memoryKey = accessValueInMemory(getIndirectDirection(memoryKey))
    section = getSection(memoryKey)
    if (debug):
        print "SET = assigning value in section: ", section
    if (section == 0):
        memory[section][memoryKey] = value
    else:
        if (len(memory[section]) == 1):
            memory[section][0][memoryKey] = value
        else:
            memory[section][-1][memoryKey] = value

def accessValueInMemory(memoryKey):
    global memory
        ##if (isinstance(memoryKey, str)):
        ##memoryKey = accessValueInMemory(getIndirectDirection(memoryKey))
    section = getSection(memoryKey)
    if (debug):
        print "GET = accessing value in section: ", section
    if (section == 0):
        return memory[section][memoryKey]
    elif (section == 3):
        return memory[section][memoryKey]['value']
    else:
        if (len(memory[section]) == 1):
            return memory[section][0][memoryKey]
        else:
            return memory[section][-1][memoryKey]
```

Para la clase Memoria se usa la siguiente implementación de las funciones:

```
def assignValueInMemory(memoryKey, value):
    global listaMemoria
    aux1 = tipodato[int(memoryKey)/1000](memoryKey)
    listaMemoria[-1].listaMem[tipoActual][aux1] = value

def accessValueInMemory(memoryKey):
    global listaMemoria, tipoActual
    if constants.has_key(memoryKey):
        return constants[memoryKey]['value']
    aux1 = tipodato[int(memoryKey)/1000](memoryKey)
    return listaMemoria[-1].listaMem[tipoActual][aux1]
```

La sección de código donde se contabiliza el tiempo de acceso y asignación de la memoria es el siguiente en ambas máquinas virtuales. Se puede observar que corresponde a la asignación (operador =) y operadores aritméticos y lógicos.

```
def doOperation(quadruplet):
    global instructionCounter, functionDictionary, instructionStack, functionScope, totalTimeAccess
        , totalTimeAssign
    if (debug):
        print quadruplet
    if (quadruplet[0] < 10):
        t1 = timer();
        elem1 = accessValueInMemory(quadruplet[1])
        elem2 = accessValueInMemory(quadruplet[2])
        t2 = timer();
        if (quadruplet[0] == 0):
            result = elem1 + elem2
        elif (quadruplet[0] == 1):
            result = elem1 * elem2
        elif (quadruplet[0] == 2):
            result = elem1 - elem2
        elif (quadruplet[0] == 3):
            result = 1.0 * elem1 / elem2
        elif (quadruplet[0] == 4):
            result = elem1 and elem2
        elif (quadruplet[0] == 5):
            result = elem1 or elem2
        elif (quadruplet[0] == 6):
            result = elem1 < elem2
        elif (quadruplet[0] == 7):
            result = elem1 > elem2
        elif (quadruplet[0] == 8):
            result = elem1 != elem2
        elif (quadruplet[0] == 9):
            result = elem1 == elem2
        if (debug):
            print "elem1: ", elem1
            print "elem2: ", elem2
            print "result: ", result
        t3 = timer()
        assignValueInMemory(quadruplet[3], result)
        t4 = timer()
        if (timeIt):
            print "Access operands : ", t2-t1
            print "Assign result   : ", t4-t3
            totalTimeAccess += t2-t1
            totalTimeAssign += t4-t3
        return True
    elif (quadruplet[0] == 10):
        t1 = timer()
        result = accessValueInMemory(quadruplet[1])
        t2 = timer()
        assignValueInMemory(quadruplet[3], result)
        t3 = timer()
        if (timeIt):
            print "Access operand : ", t2-t1
            print "Assign result  : ", t3-t2
        totalTimeAccess += t2-t1
        totalTimeAssign += t3-t2
        return True
```

Programas prueba

Con el fin de obtener datos más confiables, cada programa se ejecutará 100,000 veces para minimizar la incertidumbre y margen de error.

Esto se aprecia en el main de ambas máquinas virtuales:

Main con clase Memoria.

```
# Main.
readRawCode('rawCode.xml')
count = 100000
while count > 0:
    memoriaMain = Memoria(0,{},{},{},{},{},{},{},{},{},{})
    listaMemoria.append(memoriaMain)
    instructionCounter = 1
    while 1:
        if (doOperation(quadruplets[instructionCounter - 1])):
            instructionCounter += 1;
        else:
            break;
    count -= 1
print "Total access time: ", totalTimeAccess
print "Total assign time: ", totalTimeAssign
```

Main con estructura de diccionarios.

```
# Main.
readRawCode('rawCode.xml')
if (debug):
    print "Initial memory: ", memory
count = 100000
while count > 0:
    instructionCounter = 1
    memory = [{}, [{}], [{}], constants]
    memory[3] = constants
    while 1:
        if (doOperation(quadruplets[instructionCounter - 1])):
            instructionCounter += 1;
        else:
            break;
    if (debug):
        print "Final memory: ", memory
    count -= 1
print "Total access time: ", totalTimeAccess
print "Total assign time: ", totalTimeAssign
```

El primer programa solo tiene operaciones aritméticas y es el siguiente:

```
Program PruebaAritmeticos:
    Main:
        var int: n;
        var int: n2;
        var int: n3;
        var int: n4;

        var int: A;
        var int: B;
        var int: C;
        n = 2;
        n2 = 4;
        n3 = 3;
        n4 = 5;

        A = (5 * n - n2 / 22) + 3 * n4 - 5 + n3 * (n3 - n4 * 22);
        B = 22 / 3 + (n * n2 / (n4 * 1 - n3 - (3 * 5 - n4) - (22 / 2)));
        C = 33 + n2 / (n2 - 3 * n3) + (n4 * 1 / n);
    end
End
```

El segundo programa es una mezcla de operadores lógicos y aritméticos, junto con estatutos no secuenciales.

```
Program PruebaCondicionalesAndOr:
    Main:
        var boolean: verdad;
        var int: n;
        var int: n2;
        var int: n3;
        var int: n4;
        var int: A;
        var int: B;
        var int: C;
        n = 2;
        n2 = 4;
        n3 = 22;
        n4 = 67;
        verdad = true;

        if (n > n3 && n4 < 5):
            A = n * n2 * n3 * n4;
        else:
            B = n2 + 33;
        end;

        if (n > n4 || verdad):
            A = (5 * n - n2 / 101) + 12 * n4 - 7 + n3 * (n2 - n4 * 22);
        else:
            if (n < 0):
                B = 2 / 3 + (n * n2 / (n4 * 1 - n2 - (29 * 5 - n4) - (34 / 2)));
                end;
            A = (5 * n - n2 / 22) + 3 * n4 - 3 + n3 * (n3 - n4 * 11);
            B = 22 / 34 + (n * n2 / (n4 * 1 - n3 - (3 * 5 - n4) - (92 / 2)));
            C = 33 + n2 / (n2 - 3 * n3) + (n4 * 1 / n);
        end;
    end
End
```

Resultados:

Programa 1

```
[Bernardos-MacBook-Pro:Efficiency Test bernardot$ python arcadameVM_memory.py
Total access time:  1.22041559219
Total assign time:  1.73637533188
[Bernardos-MacBook-Pro:Efficiency Test bernardot$ python arcadameVM_dictionaries.py
Total access time:  1.13546538353
Total assign time:  0.984332084656
```

Programa 2

```
[Bernardos-MacBook-Pro:Efficiency Test bernardot$ python arcadameVM_memory.py
Total access time:  1.35263061523
Total assign time:  2.16589975357
[Bernardos-MacBook-Pro:Efficiency Test bernardot$ python arcadameVM_dictionaries.py
Total access time:  1.15610384941
Total assign time:  0.994514465332
```

Conclusiones

En base a los resultados se puede ver una eficiencia mucho mayor en la máquina que usa una estructura de diccionarios como memoria de ejecución. Para corroborar los resultados, se corrió el programa varias veces y se obtenían resultados similares. De esta forma se rompe un poco el mito de que los diccionarios es una estructura poco eficiente o lenta. La verdad es que en Python, según varias fuentes consultadas, el diccionario es una estructura a la que más se ha esforzado por amortizar los “worse cases” para hacerlos más eficientes. Es una estructura que hasta es utilizada internamente para el manejo de las clases. Por tal motivo, se esperaría que fuera eficiente en la lectura y escritura por medio de su función de hash, y este pequeño experimento lo confirma.

El experimento, los programas utilizados, máquinas virtuales, estarán en https://github.com/bernardotc/Arcadame en la sección de Efficiency Test para el que quiera repetir el procedimiento.

Referencias

- https://wiki.python.org/moin/TimeComplexity
- https://www.safaribooksonline.com/library/view/high-performance-python/9781449361747/ch04.html
- http://www.monitis.com/blog/python-performance-tips-part-1/
- http://stackoverflow.com/questions/1418588/how-expensive-are-python-dictionaries-to-handle

Agradecimientos

[1] Un agradecimiento a Maribel Pastrana Brito y a Norma Elizabeth Morales Cruz por compartir con nuestro equipo el código de su clase Memoria para ser usado como referencia y punto de comparación en este experimento con fines académicos e informativos.