

CMSC 31100: Reduce, Reuse, and Recycle: approaches to boosting memory system performance.

Lecturer: Anne Rogers

The three things to talk about:

1. Reduce: MST, VF
2. Reuse data: register allocation, caches/Virtual Memory
3. Recycle space: compacting garbage collection

Minimum Spanning Tree (MST)

A *Spanning Tree* is a subgraph that contains all the vertices and is a tree. If the weights are assigned to the edges of the ST then it is reasonable to look for a *Minimum Spanning Tree*, a tree having the minimum sum of the weights of all edges.

How to find MST?

Kruskal's algorithm

The idea is to add every new edge with a minimum weight to the current MST unless the edge doesn't make a cycle. The algorithm uses the notion of the *connected components*.

1. Put every vertex in its own set.
2. Sort the edges in a non-decreasing order, call this set as SE.
3. for all (u,v) in SE {
 if set(a) != set(b) then
 Union(set(a), set(b));
 }

The weakest place in this algorithm is the test of two sets whether they are disjoint or not. So, to optimize this part of the code it is helpful to use technique called *Disjoint Set Union*.

Disjoint Set Union

1. makeset(x) : $P[x] = x$ – initially a parent of the vertex is the vertex itself
2. union(x,y) = link(find(x),find(y)),
 where find(x) : if ($P[x] \neq x$) return find($P[x]$);
 link(x,y): $P[x] = y$ or $P[y] = x$, if x,y – roots

Algorithm cost $\sim O(n^2)$, n – number of the vertices of the tree.

An optimization trick can be done by balancing the tree. For this we use so called ranks of the vertices. A *rank* of a vertex is the number of its “children”.

```
if rank(x) > rank(y) {  
    P[y] = x  
    rank(x) = rank(x)+rank(y)  
else  
    P[x] = y
```

```

    rank(y) = rank(x)+rank(y)
}

if rank(x) = rank(y) {
    P[y] = x
    rank(y) = rank(y) + 1
}

```

It is clear that if the branch of a tree is long enough then we will spend a lot of time on finding the root of each vertex in this branch. One way to solve this problem is to use *path compression*, when the pointers between the vertices along the given branch are broken and rearranged so that each of these vertices points directly to the root.

ex: before the compression: $x \rightarrow y \rightarrow z$ (long branch)
 after the compression: $x \rightarrow z \leftarrow y$ (x,y both point to z)

And this is worth doing since the cost is approximately $n \cdot a(n)$, where $a(n)$ is comparable with $\log(n)$.

Reuse data

A simple memory hierarchy is as follows:

CPU \rightarrow L1 \rightarrow L2 \rightarrow Virtual Memory \rightarrow Disk (Physical Memory)
 3 cycles 14 cycles >100 cycles
 caches

Memory Hierarchy

fast	<----->	slow
expensive		cheap
small size		large size

Register allocation

The problem is to decide whether there is enough registers to make computations of a particular algorithm. To answer this question, *data flow analysis* of the algorithm is performed.

According to this analysis, the algorithm is divided into *basic blocks*, parts of the program that have no exits in between the code within a block. Then for each block one looks for a *live* variables, the variables that can be used later after this block. After verifying them, a graph of their relations is constructed, that is, two nodes (variables) are connected if they both appear in any block as the live variables. Once a graph is constructed, then it suffices to find such a coloring of it into n colors (= number of registers) that no vertex has more than $n-1$ edges colored by the same color.

The existence of the coloring proves the sufficiency of the registers to perform the required computations, and, moreover, gives the way to allocate them. This algorithm is called *graph coloring*.

Dynamic techniques

Important questions

1. How do I know that the data I want is in the memory?
2. If it's there then where is it in the memory?
3. What is the replacement policy?
4. How do I write new data?

These questions have different answers in different levels of memory hierarchy.

Virtual Memory

- dynamic RAM
- large additional memory
- on demand disk memory access
- locality: the objects that are close together are used together frequently
 - spatial: pages
 - temporal: array, loop index variable
- long cycles
- slow access
- cheap

How does it work?

The idea is to map Virtual Address Space into Physical Address Space, that is, basically, we perform address translation between these two space and verify whether a desired data is in the physical memory.

Virtual Address (VA) = Virtual Page Number + Virtual Page Offset

Physical Address (PA) = Physical Page Number + Physical Page Offset

Address translation

VAS		VPN + VPO
	- Page tables	
PAS		PPN + VPO

where a **page table** is the data structure used by a virtual memory system in a computer operating system to store the mapping between virtual addresses and physical addresses. (http://en.wikipedia.org/wiki/Page_table)

Replacement policy: Least Recently Used (LRU)

ex: Assume we have next sequence of block: 7 6 5 5 3 2 1 7 9 4
if we add 9th block then we throw 6th block (it is LRU)

So, new data is written over the LRU block.

Cash

- static RAM
- small size
- temporal locality
- short cycles
- fast access
- expensive

Slot – place in a cash

Block – data

Cash size = 32 bytes

Block size = 4 bytes

#Slots = 8

How does it work?

Again, having a virtual address, we find corresponding block in a cash and check for the presence of data.

VA = tag (6 bits) + offset (2 bits)
tag = block #

Replacement policy: “kick out what is there”

Types of organization: directed mapped cash/ set associatives

In the directed mapped cashes:

VA = tag (3 bits) + index (3 bits) + offset (2 bits)
index - # of block

Common problem is conflict misses, when two pick go to 1 slot. To avoid these set associative organization is mostly used.

In the set associative structure the cash is divided into pairs of slots, so that the reference to one set of two different picks don't cause conflict “which one to write?”, but can store both of them.

VA = tag (4 bits) + index (2 bits) + offset (2 bits)
index - # of set

Replacement Policy: LRU, random

Locality in code

example 1. x[10][10]

```
for (i=1,i<=10, i++) {  
  for (j=1,j<=10, j++) {  
    x[i][j] = f1; // good locality, scans whole row at once and goes to the next row  
    x[j][i] = f2; // bad locality, jumps from row to row, returning back again  
  }  
}
```

example 2. y[large]

```
for (i=1,i<=n, i++) {  
  for (j=1,j<=large, j++) {  
    y[j] = f(y[j]); // bad locality, many returns to do one operation  
  }  
}  
  
for (j=1,j<=large, j++) {  
  for (i=1,i<=n, i++) {  
    y[j] = f(y[j]); // good locality, does all the operations at once, then goes further  
  }  
}
```

To achieve better performance, it is useful to divide the structure into block, do all the computations there and go to the next one. Another way is to reorganize the loops.

Garbage Collection

Semispac compacting collection

The memory is divided into two parts labeled “To” and “From”. “To”-part is used to allocate the memory until it runs out of space. When it happens, it traces through all of the roots of the program, copying blocks to “From”-part. After this, two parts switch the roles: “To” becomes “From” and “From” becomes “To”, that is, the part that was active to be used becomes now passive and vice versa.