CS590D Presentation:

Texas Persistent Store — Pointer Swizzling At Page Fault Time

Presented by
Changgui Shi

Nov. 12, 1996

References

1. **Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson,** *Texas: An Efficient, Portable Persistent Store*, **CS Dept., The University of Texas at Austin.**

2. **Paul R. Wilson and Sheetal V. Kakkad,** *Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Address Spaces on Standard Hardware*, **CS Dept., The University of Texas at Austin.**

3. **Texas source code package**

Overview

- **Texas is a high performance persistent object store**
- **Texas was implemented as a C++ library**
- **Texas object persistence is orthogonal to its type**
- **Texas manages both transient objects and persistent objects**
- **Texas supports huge address space – 64 bits**
- **Texas uses virtual memory as caching**
- **Texas supports logging and recovery**
- **The key feature of Texas is**
  pointer swizzling at page fault time

Texas System Features

- **Transparency:** A program accesses transient and persistent objects in the same way
- **Efficiency:** In most case, access to persistent objects is as fast as access to transient objects
- **Performance:** Repeated touches to a page incur no extra overhead
- **Robustness:** Texas supports logging and recovery
- **Portability:** Texas can be used with most C++ Compilers and modern operating systems
- **Compatibility:** Compatible with existing code libraries; Minimal source code modifications are neccessary for a C++ program to take full advantage of Texas persistent storage and recovery facilities
- **Pay-as-you-go:** Pointer swizzling costs are incurred only by programs that need them, rather than by all programs
- **Modularity:** Texas is composed of largely orthogonal modules, as see in Figure 1.

Pointer Swizzling At Page fault Time

- Identifying persistent objects by reachability?
- Pagewise file management
- Using OS virtual memory management facilities to allocate, protect, unprotect or fault pages
- Texas uses virtual memory pages as cache
- Pagewise pointer swizzling

Algorithm

- When a root object is requested, Texas reserves and access protects a virtual memory page as a placeholder for the persistent object;
- When a program accesses an object in a protected page, Texas faults the page into memory. Then all pointers in the page are scaned and swizzled (pages are reserved and access protected);
- The way of incremental pagewise wavefront of faulting-and-reserving is used to access other objects. See Figure 2 and 3;
- Pointers in previously touched pages can be dereferenced just as they are in normal virtual memory pages.

Advantages

- High performance: One page fault can be shared by many other accesses
- Handling the pointer size mismatch: using large persistent address space, 64 bits for persistent address, 32 bits for virtual memory address
- Using existing compilers
- Sharing pages whithin one node or across network is possible
- Compatible with existing binaries

Problem

- Problem: Exhausting the virtual memory address space — If too many pages are reserved, eventually there will be no more uncommitted pages of address space, and the pointers to new persistent pages cannot be translated.
- How to solve the problem:
  1. Reusing virtual memory address space
  2. Reducing the effective page size
- Rebuilt of page mapping can be done in a less expensive manner

Handling Large Objects

- **Objects that are larger than the heap page size need contiguous pages of virtual memory**
- **Texas maintains a multipage map of large objects for each persistent store**
- **The pages containing a large object are all reserved at object fault time, but each individual page is loaded when it is touched**
- **Potential problem — too many pages could be reserved, cost address space and CPU time if only small part of large object is actually used.**

Typy Descriptors

- **To locate the pointers within a fault page, Texas first finds the objects within the page, then uses** *runtime type identification* **information in the objects' header fields to identify the pointer fields**
- **The** *runtime type information* **is obtained from GNU C++ 2.2 compiler (See the Figure from Texas package)**

Heap Management

- **Texas supports multiple persistent stores, each with its own heap**
- **A heap is divided into pages, each page holds objects belonging to one persistence store**
- **Free heap space is kept in a** *list*, **which is stored in its object file**
- **Each page is splited into uniform-size chunks**
- **A large object is stored in several pages**

Caching

- **Texas uses virtual memory pages as caching buffer**
- **A swizzled page can be paged in and page out**
- **Texas can be modified to evict pages back to persistent store**

Storage Management and Recovery

- Current Texas implementation uses write-ahead logging
- Texas authors proposed log-structured storage system, LSS
- LSS file stucture, see Figure 4 and Figure 5
- LSS makes checkpointing and version management easy
- Log reclamation — matter of garbage collection

Sub-page Logging

- Why ? — pagewise operation could make checkpointing expensive. If transactions have poor locality of writes, pagewise checkpointing will save too much unchanged data; single write to a page during a transaction will cause the entire page to be written to disk.
- diffining — word-by-word comparing a dirty page with a clean page, only write the modified bit to log page
- Time cost — page diffing is not too expensive
- Space cost— page diffing space cost is ok for short transactions, and can be limited by using a small LRU buffer for long transactions

An Application Example

```
#include <stream.h>
#include "texas.hh"
class prime {
    public:
       prime (unsigned long i) { number = i; next = NULL; }
       unsigned long number;
       prime *next;
  };
main (int argc, const char **argv)
{
  TEXAS_MAIN_INIT (argc, argv);
  PStore *ps = open_pstore ("primedb.pstore");
  int i = 0; prime *head;

  if (!is_root (ps, "HEAD"))
    add_root (ps, prime, "HEAD", pnew (ps, prime) (2));
  head = get_root (ps, prime, "HEAD");
  prime *ptr; unsigned long n;

  for (ptr = head; ptr; ptr = ptr->next) {
    cout << ptr->number << ' ';
    n = ptr->number;
  }

  i = 0; cout << "New primes:\n";
  for (++n; 1; n++) {
    for (ptr = head; ptr->next; ptr = ptr->next)
      if ((n % ptr->number) == 0) goto bottom_of_loop;
    ptr->next = pnew (ps, prime) (n);
    i++;  cout << n << ' ';
    if (!(i % 8))   commit_transaction (ps);
    bottom_of_loop:;
  }
  close_pstore (ps); TEXAS_MAIN_UNINIT ();
}
```

Conclusions

- Related work: ObjectStore, PS-Algol, etc.
- Pointer swizzling at page fault time is an attractive way to implement high performance persistent object store
- OO1 benchmark shows Texas has good performance
- Texas performance better than ObjectStore and E
- Texas is surprisingly simple and flexible
- Not fully implemented — no LSS, no client/server

Discussions

- Texas's good performance comes from mulitple accesses to objects and use large hardware memory, the size of virtual memory space and the size of persistent object space are not the only factors
- Pointer swizzling performance depends on applications
- Debugging persistent pointer behavior could be a nightmare
- Pagewise operation has both advantages and disadvantages