Sharing is Caring:

A Distributed Shared Memory System in User-Space

Overview

Sharing is Caring, known as SIC, is a DSM that is based on the TreadMarks paper and therefore implements a lazy release consistency model DSM. The goal of this DSM is to give userspace programs mostly transparent access to as many processors on as many computers as you like. The SIC library provides a basic interface that allows an arbitrary c program to allocate shared memory, and coordinate modifications through locks and barriers. In addition to basic tests of functionality, we have written two computationally intensive applications: a monte carlo simulation of PI and a program for locating prime palindromes in a 10 million integer range utilizing the sieve of Eratosthenes. Together, these programs provide a reasonable demonstration of the functionality of our distributed shared memory system.

Programming Model

SIC relies on the user space program to enforce consistency, providing barriers and locks to allow them to do so. Internally SIC relies on a coordinating master server to ensure that client machines acquire/release locks, and reach barriers in a consistent fashion. The master server can be run on any network-connected server, but in practice, it is collocated on one of the hosts running client code.

We chose a lazy release consistency, which means that we only tell clients to send diffs to the master on releases of locks and we only send diffs to the clients on acquires of locks. Between a lock acquire and lock release, writes are purely local, and can **only** be learned of by other nodes at one of the synchronization points. This type of model is nice for high computation, low memory footprint applications because it allows you to do computation in parallel, sync up the shared memory structure, and continue doing parallel computation. This does place the limitation that we cannot gain more memory than the smallest machine on your network has, as all shared memory could become locally mapped at any given time.

Locks in this system are network spin locks, which means that on lock acquire a client asks the master to acquire the lock, and spins until the master gives it the lock. As part of giving the client the lock, the master will transmit the memory diff of the last worker to release the lock. Note that this allows you to enforce sequential execution on distributed machines, but it also means that over time machines memory maps may become inconsistent. As such we planned to implement a sync operation that would have the coordinating master sync up with clients, but we were unable to complete this due to time constraints.

Barriers are handled in a ticket style fashion -- clients send their arrival at a barrier to the server, along with any outstanding diffs they may have. The server merges these diffs into its current state, and records the clients arrival. When the server detects that all clients have arrived at the barrier, it broadcasts a “barrier-release” message to all clients, along with a new diff created from merging each of the arriving clients. The clients merge these diffs into their local state, and continue on their way. In the client code while waiting at a barrier, the clients simply spin on a “blocked variable” while repeatedly yielding to the scheduler.

Interface

The sic system has a simple interface, similar to Treadmarks, which we outline below. (See sic.h)

```
// Must be called at start of user-space programs
void sic_init();

// Must be called at end of user-space programs
void sic_exit();

// Each client is assigned a unique id -- find the id of the currently
running
// code
int sic_id();

// Returns the number of sic-clients running on the system. Constant
throughout
// an invocation of a given sic-program.
int sic_num_clients();

// Allocate a shared chunk of memory. The memory is automatically zeroed, but
// is not backed by physical memory until it's used. All clients must issue a
// call malloc -- eg. the following code is not valid:
// if (sic_id() == 0) { sic_malloc(100); }
//
// sic_malloc involves an implicit barrier under the hood to facilitate
pointer
// sharing.
void *sic_malloc(size_t size);

// free shared memory. Currently a NOP.
void sic_free(void *ptr);

// Blocks until all calling processes arrive. All barriers must have unique
// identifiers.
void sic_barrier(uint32_t id);
```

```
// Mutual-exclusion object. All locks must have unique ids.
void sic_lock(lock_id id);

// Release mutex of given id.
void sic_unlock(lock_id id);

// Prints the current state of memory including diffs from current global
state.
void memstat();
```

Implementation

Under the hood, the memory sharing functionality in SIC is implemented with 3 linux syscalls:

- `mmap`: Reserves a predetermined amount of memory so that we can be sure that other applications will not use our VA space. Initially the entire block is marked as read only, with /dev/zero as the backing file (so that if anyone tries to read from it they get zero) on linux systems. Note that this will do on demand paging, so if we did not read protect it, we would have allocated the memory whenever we write to it. This call sets the shared base for the entire client program.
- `sigaction` to install our own segfault handler
- `mprotect` to control the writability of pages to detect and handle client writes.

Diffs are created via a comparison to a cloned copy of the page. When the client first faults on a page, we allocate a new physical page of memory and register its address. When it comes time to compute the diff, we can compare the old version to the new version to create a run-length encoded diff (see memdiff in sic-util.c). The diff is associated with the “shared” address of the page rather than the “real” virtual address the page has in memory. This allows the server to merge together multiple writes the reference the same logical region of shared memory.

When it comes time to merge diffs, we follow the path of least resistance. Simply apply all the diffs to a page containing all zeros, then diff that resulting page against 0 to create a new run-length encoded diff. This is how the server handles merging multiple client diffs for the same page.

Diffs are applied to client memory maps on lock releases and barrier releases. Because the virtual shared address is stored with the diff, and we know the shared base pointer that was returned by mmap for each client, we can easily calculate the correct offsets and make the local memory contents equal to the “agreed” upon standard of the master.

All communication is done using a custom message passing system, which in every message provides the calling client/server’s id, the type of message they are passing (e.g. CLIENT_ACQUIRE_LOCK), the value (e.g. lock # 1), and if it is an operation that requires diffs,

a diff is sent as well. Internally this is all accomplished with unix networking for tcp transport and google protobufs for marshalling.