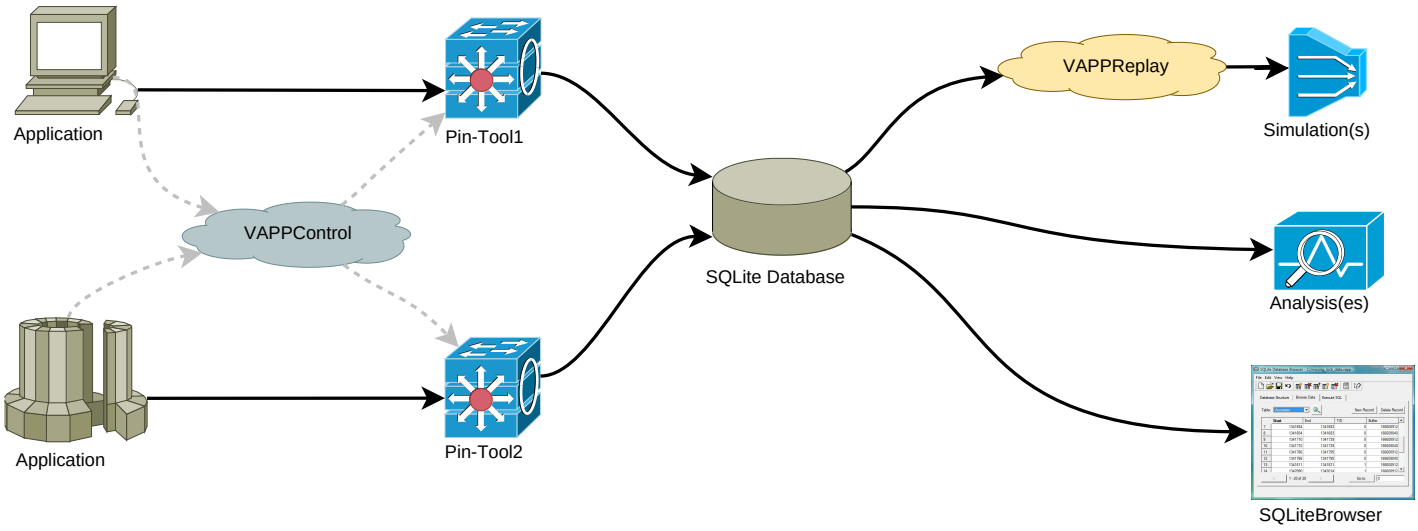
Virtual Application Profiler (VAPP)

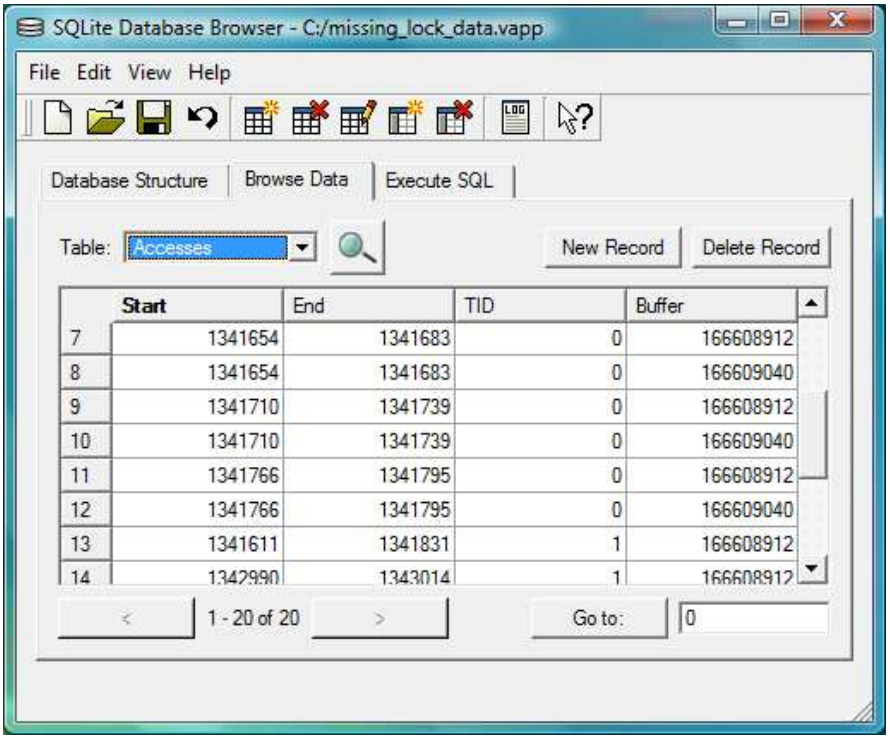
- Problem
  - Increasing hardware complexity
  - Programmers need to understand interactions between architecture and their software
- Goals
  - Quickly simulate application execution over many architecture configurations
  - Analyze and debug programs designed for parallel architecture
  - Allow developers to view and query intermediate execution traces
- Approach
  - Software-based log-and-replay system

Design



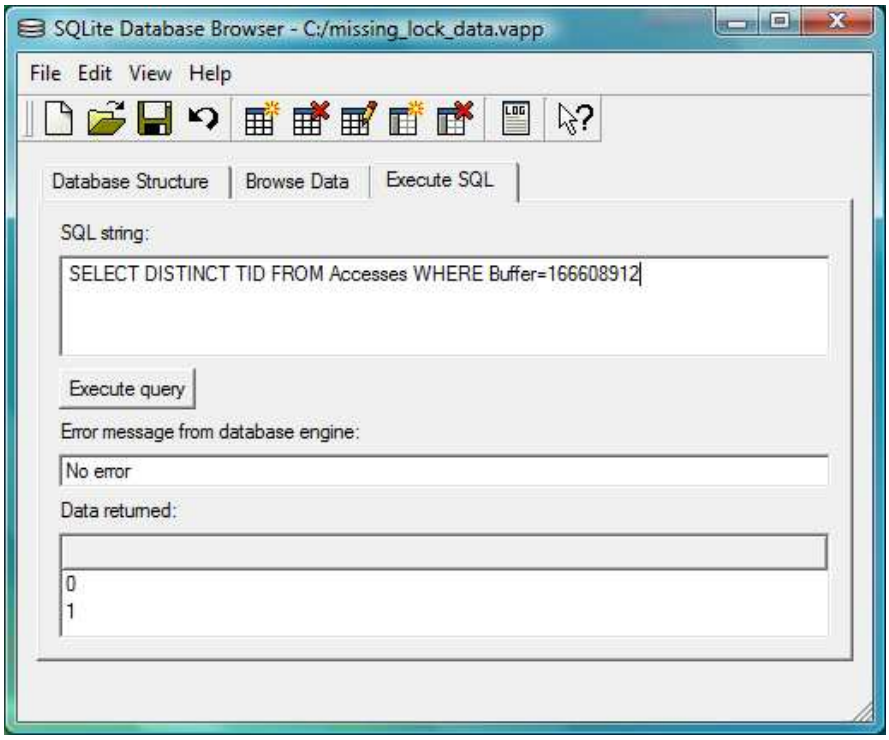
- Pin tools log memory accesses, function calls, image loads, locking, memory allocation, etc.
- SQLite database store execution trace
- Collection of replay and analysis programs take database as input
  - Replay execution on simulated hardware
  - Analyze memory access patterns
  - Lockset verification

SQLite Database



- Developer can easily examine all trace data
- SQLite Database Browser

- Many types of custom analysis can be formulated using SQL



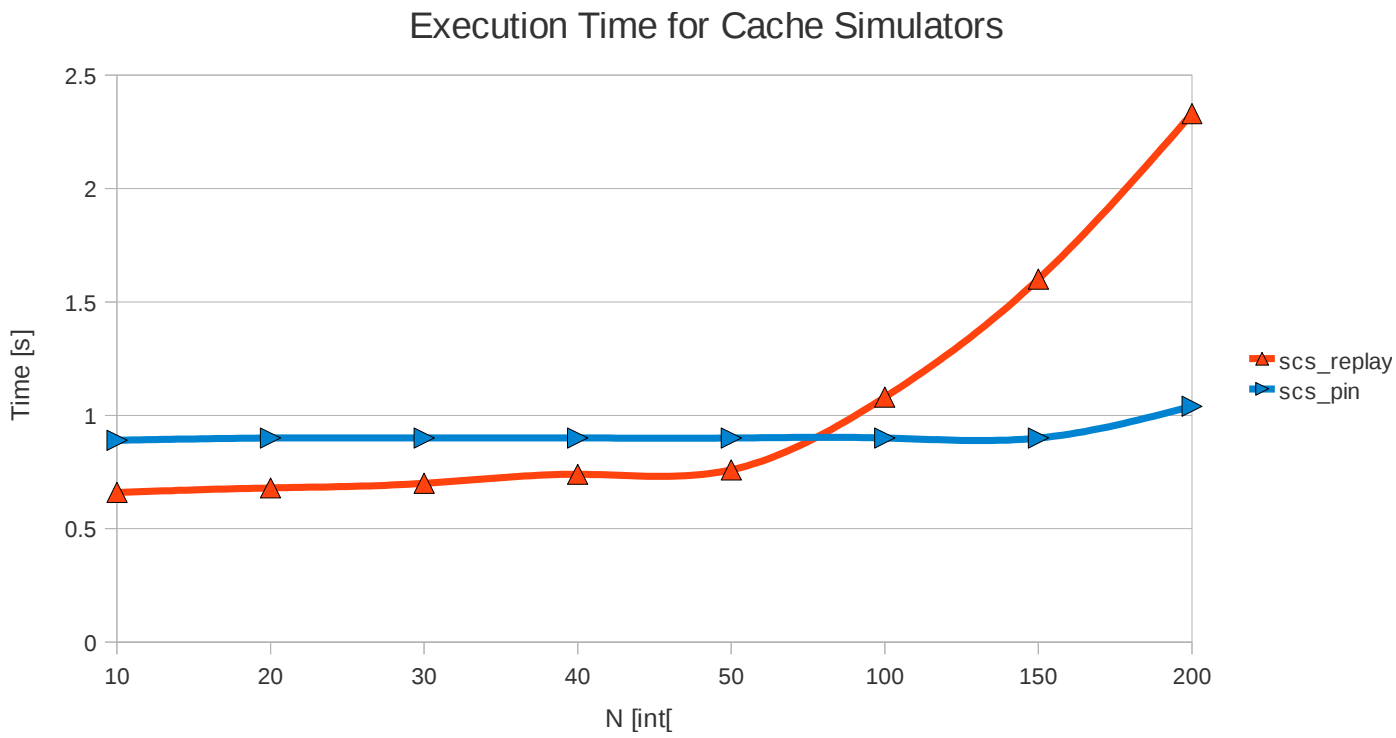
- E.g. which threads access a particular memory buffer

Replay and Analysis

- Variety of analysis tools have been implemented on top of the database
- Cache simulator
  - Prototype of a complete memory hierarchy simulator
- Lockset algorithm
  - Detect which locks protect which memory buffers in a parallel program
  - Identify inconsistent locking patterns and unprotected accesses

Simulator Performance

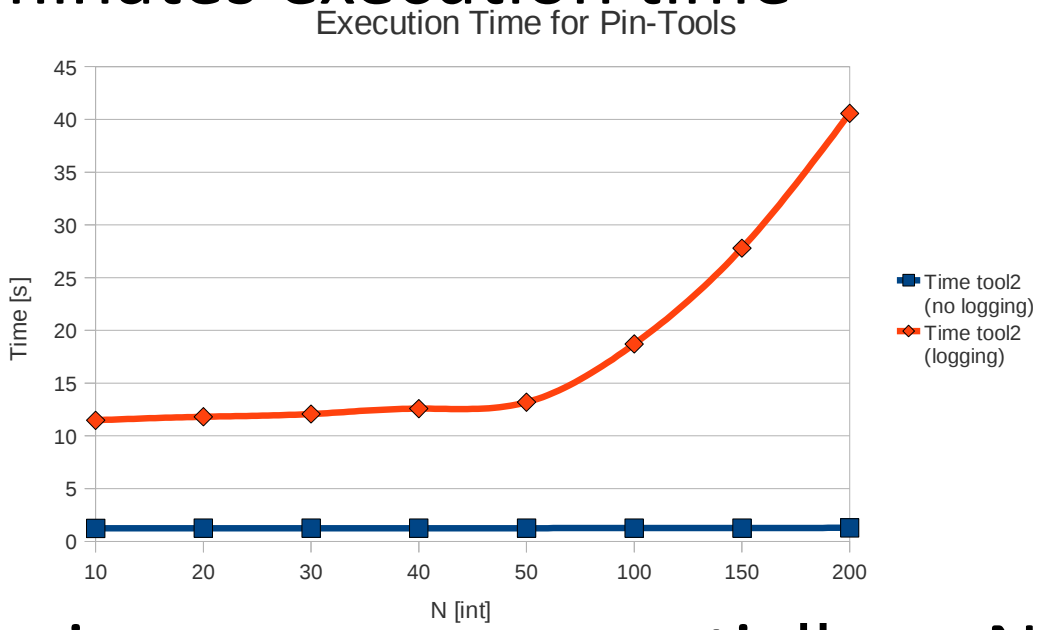
- Goal is to amortize cost of running instrumented code over many simulations
- Increase # loop iterations (N) in simple benchmark application



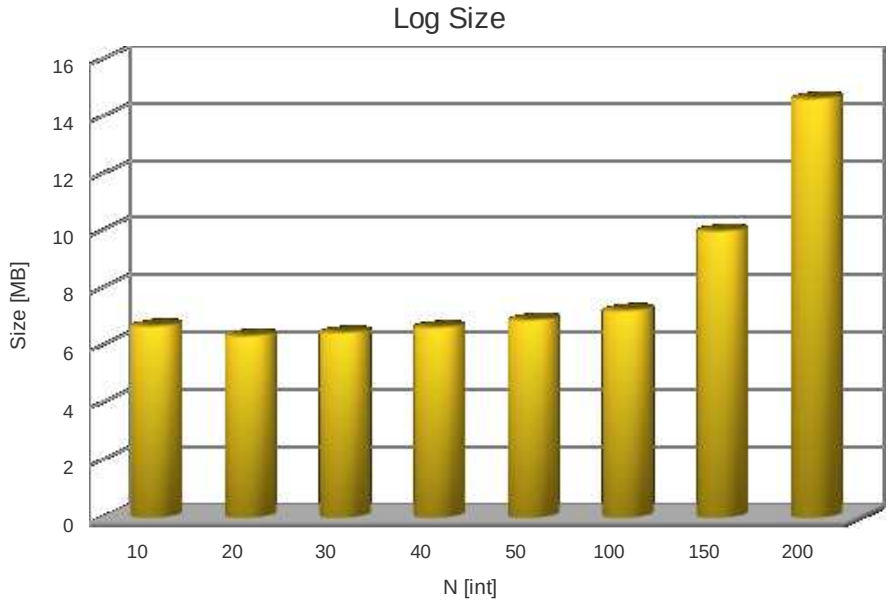
- In practice, prototype version is bogged down by I/O costs of database
- For large applications, running pure Pin versions is better (for now)

Logging Performance

- Writing logging information dominates execution time



- Log size grows exponentially as N increases



- Allow application to dynamically enable/disable logging

Parallel Program Analysis

- Memory accesses in parallel programs tracked at buffer level
- Support pthreads and Open MP
- Discover unprotected accesses to shared memory
- Example:

```
int main(void)
{
  int *result = (int*)malloc(sizeof(int));
  int value = 3;

#pragma omp parallel for
  for ( int i = 0 ; i < 100  ; i++ )  {
    if ( i % value == 0) {
      if ( *result < i ) {
        *result = i;
      }
    }
  }
}
```

```
== ANALYSE ======================================
Thread 0 -> [11018256,11018288,11019872,11020080]>
Thread 1 -> [11018256,11018288,11019872]>
shared buffer 11018256
  t1 []
  t2 []
shared buffer 11018288
  t1 []
  t2 []
shared buffer 11019872
  t1 []
  t2 []
```

`result` **NOT protected**

```
int main(void)
{
  int *result = (int*)malloc(sizeof(int));
  int value = 3;

#pragma omp parallel for
  for ( int i = 0 ; i < 100  ; i++ )  {
    if ( i % value == 0) {
      #pragma omp critical
      {
        if ( *result < i ) {
          *result = i;
        }
      }
    }
  }
}
```

```
== ANALYSE ======================================
Thread 0 -> [32243728,32243760,32245344,32245552]>
Thread 1 -> [32243728,32243760,32245344]>
shared buffer 32243728
  t1 [666]>
  t2 [666]>
shared buffer 32243760
  t1 []
  t2 []
shared buffer 32245344
  t1 []
  t2 []
```

`result` **protected** by Internal Open MP lock

Future Work

- Implement trace compression
  - Will greatly reduce log size
  - Database I/O dominates execution time so performance will significantly improve
- Allow analysis of program behavior over multiple executions
- Track lock contention and feed information back to system
  - Recommend particular lock types
  - Pin memory that is used exclusively by one thread
- Use logged information to support deterministic replay