Virtual Application Profiler

Anthony Gitter Sven Stork

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213, USA
{agitter,svens}@cs.cmu.edu

Abstract

As hardware grows more complex and with the transition to multicore systems, it becomes increasingly important for software developers to understand the interactions between their software and the underlying architecture in order to maximize performance and ensure correctness. We present the Virtual Application Profiler, a replay and analysis tool that facilitates this understanding by allowing programmers to iteratively simulate application execution on many architecture configurations and evaluate memory usage and protection in parallel programs. Our profiler is unique in that it logs all application execution traces in a SQLite database, which allows developers to view and manipulate the logged data with ease.

General Terms profiling

Keywords profiling concurrency simulation

1. Introduction

The Virtual Application Profiler (VAPP) was designed to give programmers greater understanding and control of their code in response to the rising complexity in architecture over the past decades. In this context, architecture complexity refers to both the complicated hardware units of a single core (such as the Pentium 4) as well as the new challenges brought forth by the rise of multicore machines.

Specifically, our goals are to (1) allow programmers to quickly simulate application execution over a large set of architecture configurations, (2) assist in the testing and debugging of programs that run on parallel architecture, and (3) do so in a transparent manner that allows programmers to easily access and query those aspects of an application's behavior that are relevant to these tasks. To achieve these goals, we have pursued a software-based log-and-replay approach, in which a trace of application's execution is stored and subsequently replayed on simulated architecture or used for other high level analysis. The logged data is stored in a manner that allows the programmer to directly inspect or make custom inquiries over the trace.

Because VAPP's broad goals encompass replay on simulated architecture and data race detection, there a large body of related work. However, similar approaches focus on only one of these two problems. MemSpy [Martonosi et al. 1992], like VAPP, is a software-based profiler designed to help developers target performance bottlenecks due to the memory hierarchy. MemSpy performs more detailed analysis than what VAPP presently offers and reports statistics (such as memory stall time) at the level of code-data pairs. For instance, it can show how many stalls are due to memory accesses to a particular matrix in a specific inner loop of a program. Unlike VAPP, MemSpy performs all analysis dynamically, must be rerun every time the simulated architecture's parameters are changed, and does not concern itself with parallel program correctness. SIGMA [DeRose et al. 2002] is more similar to VAPP in that it instruments code, stores an execution trace, uses the trace to simulate execution on various hardware configurations, and logs statistics for the user to view. It is also software-based and allows users to simulate execution on many architecture configurations from a single trace file. SIGMA is also more fully featured than VAPP's simulated replay at the moment because it simulates a larger portion of the memory hierarchy, but it does not target issues related to data races and parallel program debugging.

Related work relevant to VAPP's second goal includes data race detectors and deterministic replay tools. RecPlay [Ronsse and De Bosschere 2000] is a software tool that detects data races by making multiple executions of the target program, performing different levels of analysis with each pass. This enables it to trace a minimal amount of information as opposed to all memory accesses. VAPP only requires an application to be executed once, and is suitable for more general parallel program analysis than data race detection. Eraser [Savage et al. 1997] is also software-based and implements a very similar locket algorithm to detect inconsistent lock usage in parallel programs. It performs dynamic analysis as opposed to VAPP's log-and-replay approach, and it limited to analysis of pthreads locks unlike VAPP, which also supports Open MP. BugNet [Narayanasamy et al. 2005] and FDR [Xu et al. 2003] both log program execution in order

to replay it in a deterministic manner to assist in debugging following a crash. They create checkpoints of system state and then track all execution for a fixed window (e.g. 1 billion instructions). These methods both require hardware support. FDR supports full system replay, whereas BugNet (like VAPP) focuses on user-level application code only. BugNet is unique in that it primarily records only loads, and can reconstruct deterministic replays from the load data and checkpoints. However, both FDR and BugNet do not detect data races or concurrency bugs unless these bugs cause an application or system to crash. They also can only replay a fixed window of execution, unlike VAPP which aims to replay the entire execution. Nevertheless, these two methods support a much more comprehensive form of replay than VAPP in its current state. All of the above methods differ from VAPP in that they only consider a single architecture configuration during replay and analysis. Please see Xu et al. [Xu et al. 2003] for a more comprehensive comparison of related data race detection and deterministic replay tools.

Note that none of these methods allow users to access or query information concerning application execution and behavior as VAPP does. Thus, our third goal is one of the primary novel technical contributions of this work that distinguishes our approach from the multitude of related profilers and replay tools. Unifying our first and second goals in a single tool is another unique aspect of VAPP, and to our knowledge all existing related work concentrates on only one these two problems that result from architecture-complexity.

We have released VAPP as an open source project, and it is available for download from our Google Code page (`http://code.google.com/p/vapp/`), which is linked from our project web site.

2. Design

Here we present the VAPP's structure and implementation details, as well as the history and reasoning that led to this final approach. Our design assumes a shared memory multiprocessor system and does not require hardware support.

2.1 VAPP Modules

The three primary layers of VAPP are a Pin[1] tool for dynamic binary instrumentation, a SQLite[2] database to store application traces, and a collection of replay and analysis programs. In addition, we have written two libraries that are used for logging control and core replay functionality. These components and their relationship are shown in Figure 1.

2.2 Pin Tool

While Pin is very feature-rich in terms of the kinds of information it provides access to during instrumented application execution, this comes at a cost with regard to execution time. Given that we wished to design a tool that would allow programmers to repeatedly run and rerun their applications on a large set of architecture configurations, one design goal was to make this process as fast as possible. We assume that the application will be executed on very many types of simulated architectures, and thus concluded that the best way to reduce the total time to conduct the simulations would be to log an execution trace and amortize the slowdown from the Pin tool over many replays. Note that this approach further assumes that the application designer wishes to use the same set of input for all replays or sufficiently large groups of replays.

To be precise, we have implemented two distinct Pin tools to focus on the detailed logging required for application replay and the different trace requirements needed for the analysis of parallel program correctness respectively. In theory, these two tools could be merged and log the union of all data that each individual tool tracks. However, we discovered that from a practical standpoint such an approach could be seriously inefficient, in particular when the user desires some form of analysis that does not require full application replay. That is, in order to replay application execution, it is necessary to log all memory accesses an other information at a very detailed level. However, our analysis of parallel programs can be achieved with a much more coarse execution trace and logging less results in faster execution and perhaps more importantly much smaller logs. Smaller log files in turn lead to faster database accesses and faster execution of our analysis programs.

As mentioned above, the replay-oriented pin tool must log memory operations in great detail, which conflicts with our desire to keep the log database as small as possible for performance reasons. Therefore, our final design allows the application to dynamically turn selectively switch certain subsets of the logging on and off or entirely disable the trace some of the execution. This is achieved via simple calls to our VAPPControl library, which only requires the application code to specify which logging features are to be enabled or disabled at that point. The Pin tool instruments such library calls and then adjust the logging accordingly. The specific trace information that can be controlled in this manner is function calls, memory accesses, memory allocations and deallocations, and locking and unlocking.

The parallel program-oriented tool is able to track execution as a coarser grain by analyzing memory operations at the level of buffers instead of individual addresses. We define a buffer as the block of memory that is allocated by a single call to `malloc`. Thus, all parallel program correctness analysis, such as lockset analysis, is also performed at the level of buffers. Consequently, programs that lock memory at a finer level, for instance by having different locks to protect different rows or blocks of a matrix, can lead to false positives in the analysis routines. Given the performance benefits, we feel that this is a reasonable tradeoff.

[1] http://www.pintool.org/

[2] http://www.sqlite.org/

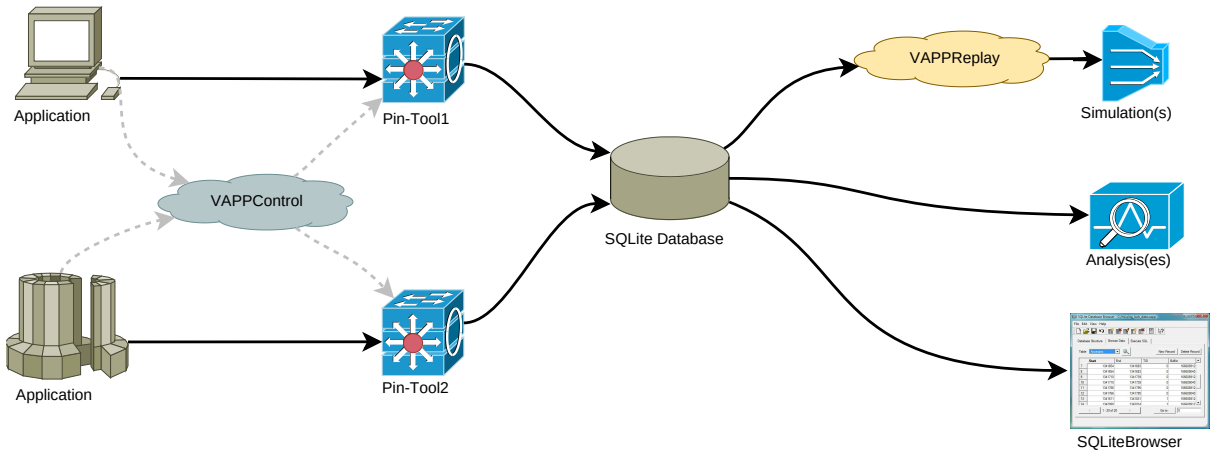

Figure 1. Applications of different platforms can be analyzed by any of our Pin tools. Using our VAPPControl library applications can limit the gather trace information in scope and time. Each Pin tool will write its data into a single SQlite database file. This database file will then be used as source for our analysis (SQL-based or direct) and as input to out VAPPReplay library. The VAPPReplay library provides a simple C++ class abstract for the ease of developing analysis or simulations.

2.3 Database Functionality and Schema

We chose SQLite to store our application traces for its usability and portability. The entire database that stores the logs is saved as a single file that can be easily moved across machines/platforms or archived. In addition, SQLite was easy to learn and begin using, which was of critical importance given the strict timeline for our project. Moreover, the open source SQLite Database Browser[3] allows application programmers (our intended users) to easily view and query the SQLite database, which was a significant goal of our work. By exposing the logged data, developers can implement certain analysis and explore their application's behavior directly without writing custom replay code. Figures 2 and 3 provide one such an example where a user is investigating whether a particular memory buffer is accessed by a single thread.

Figure 4 shows the database schema for the first Pin tool, that records all memory operations. The schema consists of the following tables:

Images Tracks information about which images (i.e. executables and dynamic libraries) are used by the program). The table records the name (`ImgName`) of the image and an associated, unique numerical identifier (`Id`).

Calls Records all methods calls performed during execution. An entry consists of the simulation time (`VLCK`) the event occurred, the memory base address (`Method`) of the method and the action (`Enter`), i.e. whether the program entered or left the method.

Methods Contains all available methods during the execution of the program. An entry of this table consists of the Method name (`MethName`), the id (`ImgId`) of the image within which the method is located, the base address (`MethStart`) at which the method is located in memory and the address of the last method instruction (`MethEnd`).

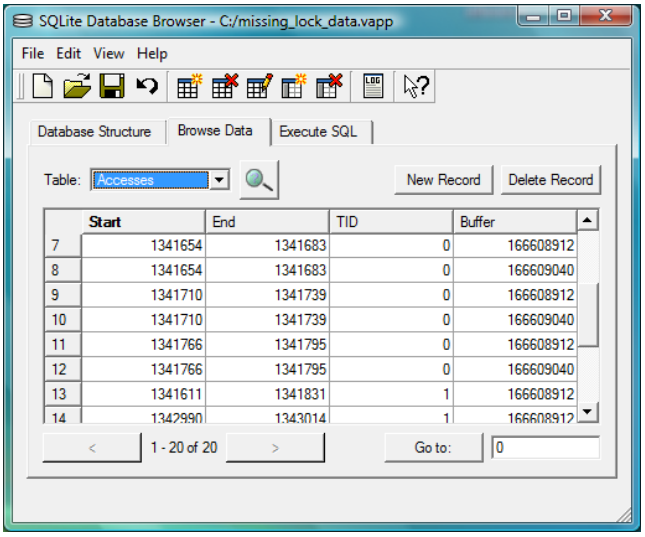

Figure 2. Storing application traces as an SQLite database allows users to easily access the data with the SQLite Database Browser.

MemoryAccesses Records all occurred memory accesses. If an instruction has more than one memory access (e.g., instruction and data access) then multiple entries are generated. An entry consists of the simulation time (`VLCK`) the event occurred, The memory address that is accesses (`MemAddress`), the address of the instruction that triggered the event (`InstrAddress`), an numerical value (`ThreadId`) specifying in which thread the event oc-

[3] http://sqlitebrowser.sourceforge.net/

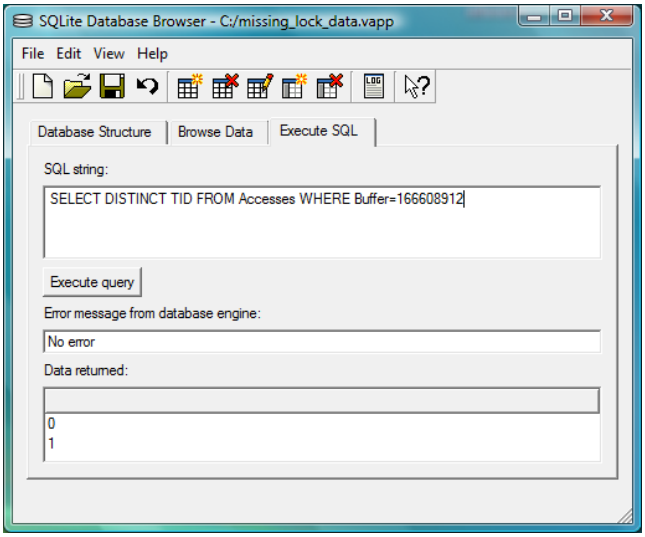

Figure 3. Users can query the SQLite database to learn about their application's execution. Here the user has determined that a particular memory buffer was accessed by multiple threads.

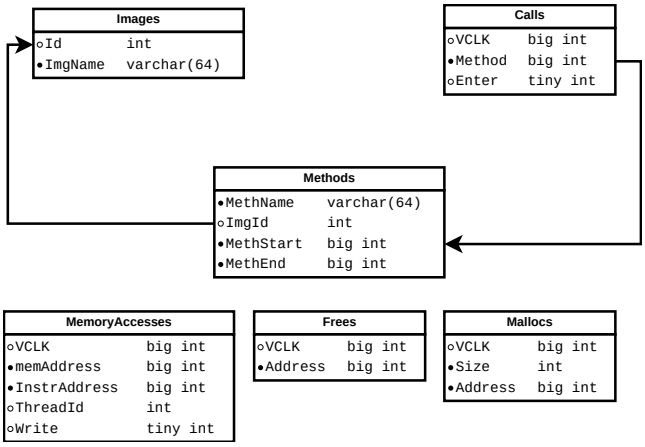

Figure 4. Database Schema Pin-Tool1

curred and field identifying (`Write`) whether the operation was a read or write.

Frees This table records all executed `free` calls. An entry in this table consists of the simulation time (`VLCK`) the call occurred and the base address (`Address`) of the buffer.

Mallocs This table records all memory allocations via `malloc`. An entry in this table consists of the simulation time (`VLCK`) the call occurred, the size of the allocated buffer (`Size`) and the base address (`Address`) of the buffer.

Figure 5 illustrates the database schema of the Pin tool designed for parallel program analysis. The schema is composed of the the follow tables:

Threads This table gather all information of the threads that have been executed during the program execution. An

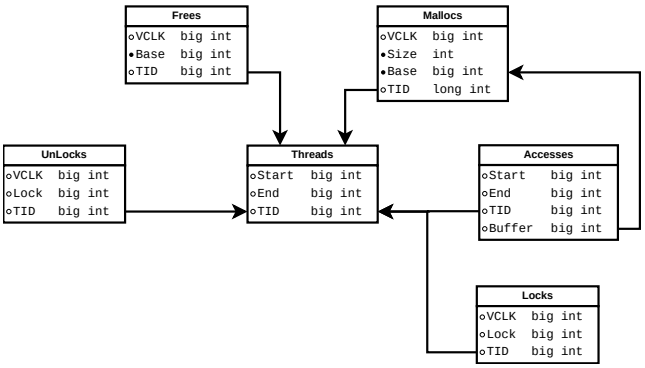

Figure 5. Database Schema Pin-Tool2

entry in the table consists of the time the thread has been created (`Start`), the time the thread terminated (`End`) and an unique,numerical identifier (`TID`).

Accesses This tables track execution epochs and the buffers touched during this execution period. To avoid variable length entries in the database we use fixed size entries to specify one buffer/epoch mapping. An entry in this table consists of the time the execution epoch begins (`Start`), the time when it ends (`End`), the number of the thread that executed this epoch (`TID`) and the base address of the buffer (`Buffer`).

Locks This table records all executed `pthread_mutex_lock` and `GOMP_critical_start` calls. An entry in this table consists of the simulation time (`VLCK`) the call occurred, the base address (`Address`) of the lock and the numerical identifier of the thread performing the operation (`TID`).

UnLocks This table records all executed `pthread_mutex_unlock` and `GOMP_critical_end` calls. An entry in this table consists of the simulation time (`VLCK`) the call occurred, the base address (`Address`) of the lock and the numerical identifier of the thread performing the operation (`TID`).

Frees This table records all executed `free` calls. An entry in this table consists of the simulation time (`VLCK`) the call occurred, the base address (`Address`) of the buffer and the numerical identifier of the thread performing the operation (`TID`).

Mallocs This table records all memory allocations via `malloc`. An entry in this table consists of the simulation time (`VLCK`) the call occurred, the size of the allocated buffer (`Size`), the base address (`Address`) of the buffer and the numerical identifier of the thread performing the operation (`TID`).

2.4 Application Replay

Presently, VAPP supports the replay of memory operations with particular focus on memory accesses and usage patterns. This level of replay is sufficient to allow a user to simulate execution on a variety of memory-related hardware components. We felt this was an important area to focus on

first because memory accesses can oftentimes be the major bottleneck in program performance. By replaying memory operations, a developer can investigate how varying levels of cache or page tables, cache configurations, shared/private caches, shared/separate L1 instruction and data cache, etc. impact performance.

The core replay functionality is implemented in the VAPP replay library. One of the central parts if the library is the Executor class (see Figure 6). The Executor class is an abstract class that provides callback methods for every memory operations. Every simulation/analysis needs to inherit from the Executor class and implement the callback methods. The programmer then can register its implementation (or several implementations if necessary or required) with the library and start the replay. The replay library loads memory operations from the database and replays them into every executor that has been attached to it.

2.5 Parallel Program Analysis

Our analysis of code running on parallel architectures is not replay-based and thus does not require individual memory accesses to be logged or loaded. Specifically, VAPP can examine shared memory protection and locking behavior via the classic lockset algorithm [Savage et al. 1997], determine which memory buffers are accessed by which threads, and report which buffers are allocated but not used or not freed.

The goal of this analysis is to allow the programmer to make educated architecture-centered decisions. For instance, if it is discovered that certain memory buffers are consistently accessed only by a single processor, that information can be fed back to the system so that this memory is pinned to that processor's cache so that communication overhead is reduced. Information regarding lock usage and contention can be used to determine what types of low-level locking mechanisms might be best for the application given the performance, communication, and space tradeoffs in these mechanisms. Lockset analysis is an fundamental component in the analysis of parallel program correctness. By reporting which memory buffers are not protected by a consistent set of locks when they are accessed, the programmer can detect concurrency bugs. Conversely, if it is discovered that a buffer is protected by multiple locks, the programmer may realize that certain locks are redundant and improve performance by eliminated unneeded locks.

We extended this analysis into a second set of analysis tools that allow the user to analyze locksets amongst several different threads. This analysis reports which buffers are shared amongst which threads along with their corresponding locking set.

While there are a number of parallel programming models in use today, VAPP focuses on the popular Open MP[4] and pthreads[5] approaches. Pthread-based parallel programs typically protect memory via explicit calls to `pthread_mutex_lock` and `pthread_mutex_unlock`, thus VAPP tracks these calls. Open MP support comes in the form of logging the use of critical sections pragmas and the memory they protect. If the Open MP implementation would use the pthread functionality then we would already have everything in place to analyze Open MP programs. Unfortunately the Open MP implementation of GCC (called GOMP) does not use pthread mutexes. After analyzing the source code of the corresponding runtime library, we discovered that the implementation uses system level locks. Fortunately the runtime implementation used a similar abstraction. Every time a critical section is entered the `GOMP_critical_start` method is called to lock a default lock, and the `GOMP_critical_end` method is call when leaving a critical section to unlock the default lock. We therefore treat these like the `pthread_mutex_lock` and `pthread_mutex_unlock` functions. Because the GOMP function always lock a default lock, there is no need to pass the lock explicitly as parameter to the function. We therefore cannot detect the lock address when those methods are called. Fortunately we do not need the exact address, because the those functions will always lock/unlock the same lock. We therefore replace the lock address in the case of GOMP function call with a magic value (in our case 666) that is unlikely to be used elsewhere.

2.6 Evolution From Initial Design

VAPP's focus has evolved since its inception. Initially, the majority of our efforts were to focus on the goal of making single-threaded applications' execution traces accessible via SQL queries, because we felt this was the most unique component of our proposed work. We also intended to support efficient iterative simulations as the current version of VAPP does. However, after helpful discussions with Dr. Mowry we came to the conclusion that this scope alone was more interesting to a software engineering audience than the architecture community. Thus, we scaled back our work on the rich analysis that could be expressed purely as SQL queries and added our parallel architecture-oriented goals as a major component of VAPP.

As discussed previously, the size of the SQLite database logs became problematic when logging detailed traces on larger applications. This led us to introduce the VAPP trace control library, which allows developers to only trace the most interested portions of an application, thereby reducing the log size. In addition, we realized that parallel programs often share data at a somewhat coarse level in order to reduce communication overhead and that logging every individual memory access may not be necessary for their analysis. Thus, we split the Pin tool so that the tool for parallel analysis could log less detailed information at the level of memory buffers. This resulted in a massive reduction in log size, which improved the performance of both the Pin tool and the analysis programs.

[4] http://openmp.org/wp/

[5] https://computing.llnl.gov/tutorials/pthreads/

```
class Executor {
 public:
  virtual ~Executor() {}

  // is called for instructions that read one memory location
  virtual void memRead(long ip, long addr, int size) = 0;

  // is called for instructions that read two memory location
  virtual void memRead(long ip, long addr1, long addr2, int size) = 0;

  // is called for instructions that write to memory
  virtual void memWrite(long ip, long addr, int size) = 0;

  // called for all other instructions
  virtual void instruction(long ip) = 0;

  // is called when the simulation ends to generate the final report
  virtual void doReport(std::ostream &os) = 0;
};
```

Figure 6. The abstract Executor class that every simulation needs to sub-class in order to receive memory callbacks.

Lastly, early on we intended to only support pthreads for our parallel analysis, but found that we had time after completing our 100% goals to extend VAPP to support Open MP as well. Due to the popularity of Open MP, this makes VAPP attractive to a much larger community of developers.

3. Experimental Setup

For our evaluation we used the latest version of the Pin tool infrastructure available (i.e. pin-2.7-29972). Unless otherwise specified, all example programs have been compiled with GCC compiler version 4.4.1, enabling debug information and disabling any optimizations. We used SQLite version 3.6.16 that was part of our default Linux distribution.

The benchmarks have been run on an Intel Core2 Quad Q6600 CPU system with 4GB of main memory, using the software packages described in the previous paragraph.

4. Experimental Evaluation

Each of our aforementioned goals has a different metric for success, thus we conducted a performance-based analysis of our simulator and replay tool and a qualitative/correctness-oriented evaluation for VAPP's parallel program analysis components. The achievement of our goal to allow users to easily view and interact with their applications' traces is demonstrated in Figures 2 and 3 and did not warrant further evaluation.

4.1 Simulator Performance

As one of our goals was to provide a fast replay mechanism for the logged data. We developed one cache simulator based on our 1^{st} assignment (hereafter referred to as *scs_pin*) and another simulator that uses our replay mechanism (hereafter referred to as *scs_replay*). Given the fact that our Pin version of the cache simulator already used the Executor abstraction, the port was straightforward and allowed us to compare functionally and implementation-wise completely identically cache simulators with each other. To evaluate the performance we wrote a simple benchmark program (see Figure 7) that initializes a memory array. We then evaluated and compared the two following approaches by varying the parameter N:

```
#define N 200

int main(void){
  int value = 0xCafeBabe;
  int *buffer;
  int i = 0;

  buffer = (int*)malloc(N*N*sizeof(int));

  for ( i = 0 ; i < N*N ; i++ ) {
      buffer[i] = value;
  }

  return buffer[0];
}
```

Figure 7. The source code for `test_init` program. (Note the memory leak is intentional.)

scs_pin Run the benchmark application with the Pin version of our cache simulator.

scs_replay First run the benchmark application with our Pin tool to gather a trace file. Then use this trace file as the source for our replay library.

Table 1 summaries the results from our evaluation. Note that there is a change of the stepping. While the first 4 runs

Program/Configuration	Time (no logging) [s]	Time (logging) [s]	log size [bytes]	scs_relay [s]	scs_pin [s]
test_init/N=10	1.23	11.48	6601728	0.66	0.89
test_init/N=20	1.23	11.48	6717440	0.66	0.89
test_init/N=30	1.25	12.07	6912000	0.70	0.90
test_init/N=40	1.24	12.57	7183360	0.74	0.90
test_init/N=50	1.25	13.19	7532544	0.76	0.90
test_init/N=100	1.24	18.71	10441728	1.08	0.90
test_init/N=150	1.26	27.79	15292416	1.60	0.90
test_init/N=200	1.28	40.57	22071296	2.33	1.04

Table 1. Performance evaluation and comparison of the cache simulator using the Pin infrastructure directly and using our replay library.

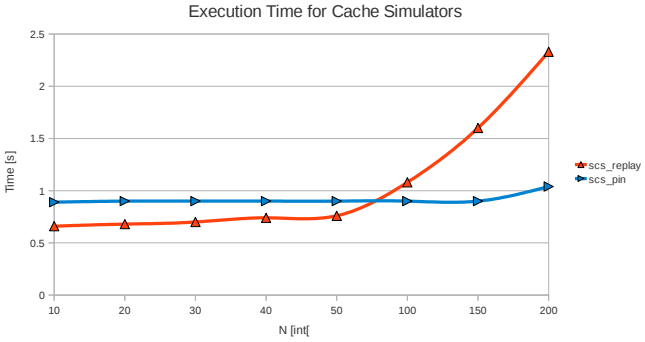

Figure 8. The execution time of the cache simulator using the Pin infrastructure or our replay library.

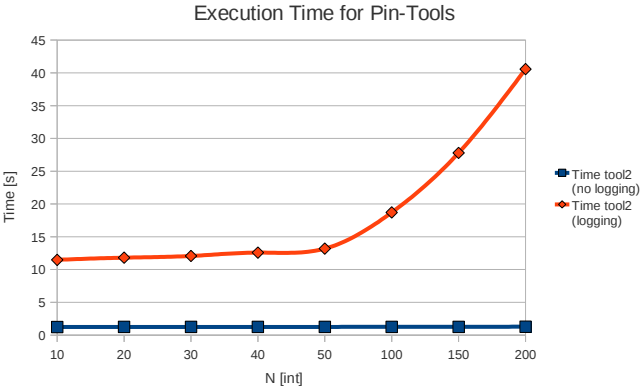

Figure 9. The execution time of our tracing generator, comparing the case where actual data is written and the case where we just run the application without performing any log operations.

increase N by 10, the last 4 runs increased N by 50. The following section will discuss these results in more detail.

As one of our main goals was to provide a faster way of performing analysis, our main interest was the comparison of the cache simulator execution times. As shown in Figure 8, the execution time for the replay version (scs_replay column in the table) is only better than for the 'native' Pin tool version (scs_pin column) for very small N. For N larger than 50, we can see a clear trend towards a drastic performance penalty for our replay approach. In the next section we will break down our analysis of this phenomenon.

While gathering the tracing information we already realized a performance hit. To get an better understanding of the possible performance costs of the SQLite database we compared the dry-run mode of our Pin tool, which does not write any logging information, with the normal mode of our Pin tool. The results are shown Figure 9. As seen in the figure, the execution time for the case without writing any logging information is almost constant as N grows, while execution with writing logging information is at least one order of magnitude worse (with a trend that appears to be exponential). A reason for this behavior can be directly inferred by observing the amount of data that is stored in the database,

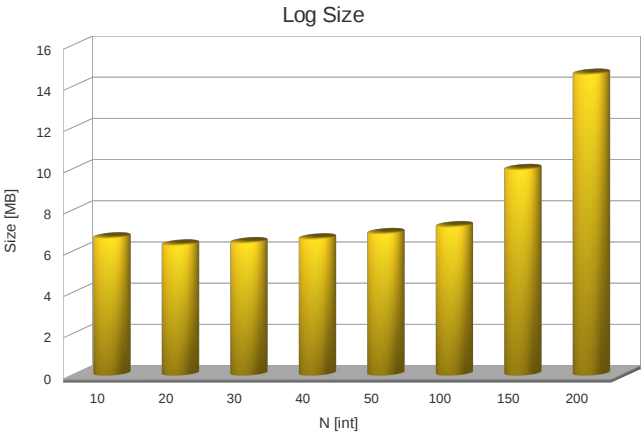

Figure 10. The growth of the log size while changing parameter N of the `test_init` program.

as shown in Figure 10[6]. The execution time of the Pin tool

[6] Note that the store operations modify a database table and do not just append the file.

is dominated by the increasing I/O costs as N grows, and the execution time and log size exhibit the same curve.

To confirm this observation and evaluate if there is a single root cause that we could potentially overcome, we recompiled the whole software stack with profiling information and re-ran the replay cache simulator with the input for N = 200. As shown in Figure 11, almost half of the time is spent by an internal SQLite function, converting the data from the database into a string representation.

To put our log size into context, we compared VAPP's logs with those from several related systems discussed in Section 1. Note that the following results were reported by the respective authors; we did not conduct these tests ourselves. SIGMA employs sophisticated compression techniques before writing its trace file, however the effectiveness of the compression is heavily dependent on the program that is executed. In some cases a several hundred MB trace was compressed to less than 1 MB, but in the worst case SIGMA's log files were several hundred MB large *after* compression. Note however that the SIGMA benchmarks were run on more complex programs than `test_init` and that VAPP's log files would likely grow to this magnitude or more if run on identical benchmarks.

FDR and BugNet log files were measured after recording a 1 billion instruction replay window. FDR's log was only 34 MB, which is vastly less than what VAPP would generated if tracing a 1 billion instruction program. However, recall that FDR utilizes advanced hardware support and compression, and also requires a full memory dump in addition to the reasonably sized log file. BugNet only logs load values, thus on average its log file was less than 20 MB during testing. In the worst case benchmark, it grew to over 100 MB. The fact that VAPP's log for a simple benchmark is roughly the same size as BugNet's for a 1 billion instruction window indicates that detailed logging requires advanced compression methods in order to scale.

4.2 Concurrency Analysis Results

As described in Section 2.5, we shifted our focus towards the analysis of concurrent programs. Here we present the a short evaluation of our second analysis. In Figure 12 a short test Open MP program is shown. While the left version of the program has a race condition, by not correctly synchronizing the access to the result buffer, the version on the right side correctly uses a critical section to protect access to the result buffer.

Below the source code of the programs the output of VAPP's extended analysis is shown[7]. First the analysis prints the threads along with their accesses buffers. Then it prints for every shared buffer the lockset of the each thread[8]. As shown on the analysis for the incorrect version, none of the 3 shared buffers are protected by a lock. In the correctly synchronized version we can see that the lockset of the first shared buffer is not empty and in fact we realize that this buffer is locked by the internal lock of the Open MP library (indicated by the magic value, as described in Section 2.5).

In both cases we see buffers that are shared and not protected by any locks. Those are internal buffers of Open MP. Given our current implementation, our analysis has the disadvantage that if a buffer is accessed without any protecting lock being held, then the resulting lockset will always be empty. This is phenomena appears if for instance one thread allocates the buffer and initializes the buffer without holding a lock. Future work could extend the current analysis to employ heuristics to detect when a buffer leaves the thread local state and enters the shared and vice versa, possibly by extending Eraser's approach to a similar problem [Savage et al. 1997]

5. Lessons Learned

One surprise we encountered relatively early in the project was the drawbacks of using a SQLite database to store the application traces. Our testing showed explosive growth of the log file size as we scaled our test programs, with some log files growing to several hundred megabytes even on moderate length programs. In addition, as seen in Section 4.1 writing to the database is a huge bottleneck in the performance of our Pin tool. While we maintain that may not be a critical problem if the user intends to amortize the initial cost of creating the trace over a very large number of simulated replays, the number of replays required to make the initial instrumented execution worthwhile is much greater than we expected. Even then we need to reduce the I/O costs, because as seen in Table 1 the amortization approach fails if a single execution of the dynamic simulator is faster than an identical simulation that reads from a trace file. This suggests that the reason previous work has not exposed the logged data to the user to the degree we do is that doing so is impractical with regard to log file size and exaggerated slowdown. Commonly used compression techniques would help reduce I/O costs and log size, but at the expense of the user's ability to interpret the stored trace.

Another unexpected challenge we faced during development was the difficulty in simulating hardware. As a 125% goal we had hoped to simulate more than just the cache, and had considered using VAPP to demonstrate the effects of using huge page tables in a system. However, we were unable to complete this task and we now understand how daunting it would be to further develop VAPP to realistically simulate a variety of real architectures and configurations of those architectures.

6. Conclusions

Modern systems present an overwhelming array of decisions to application developers who wish to optimize the archi-

[7] We ran this test on a dual core machine.

[8] Note that the addresses of the buffers may differ between different runs, depending on the address that is returned by malloc.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 43.44     0.43     0.43  4047013     0.00     0.00  sqlite3VXPrintf
 12.12     0.55     0.12   809419     0.00     0.00  sqlite3VdbeExec
  6.06     0.61     0.06   758517     0.00     0.00  Profiler::addToRead(long, long, long, long)
  3.03     0.64     0.03  5665887     0.00     0.00  sqlite3ApiExit
  3.03     0.67     0.03  4047031     0.00     0.00  sqlite3ValueText
  3.03     0.70     0.03  4047017     0.00     0.00  sqlite3StrAccumAppend
  3.03     0.73     0.03   809305     0.00     0.00  Profiler::addToUsage(long, long)
  2.53     0.76     0.03  4047023     0.00     0.00  sqlite3VdbeSerialGet
  2.02     0.78     0.02  4047031     0.00     0.00  sqlite3VdbeChangeEncoding
  2.02     0.80     0.02  4047023     0.00     0.00  columnMem
  2.02     0.82     0.02  4047012     0.00     0.00  sqlite3_snprintf
  2.02     0.84     0.02   809401     0.00     0.00  callback(void*, int, char**, char**)
  1.52     0.85     0.02  4047059     0.00     0.00  sqlite3DbMallocSize
  :
```

Figure 11. GProf results of the cache simulator run using our replay library

```
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 []
```

```
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 []
```

Figure 12. Comparison of an incorrect (left) and a correct (right) Open MP program, along with the results of our analysis.

tecture configuration for their application. Simultaneously, the rise of multicore machines and parallel architectures has further complicated this landscape, and the shift to a parallel paradigm brings forth new issues pertaining to both performance and correctness. To facilitate application development in this context, we have developed VAPP, a profiling, replay, and analysis tool that requires no hardware support and can assist in simulating program execution on different architecture configurations and evaluating parallel programs. VAPP stresses transparency, and all logged execution traces can be easily accessed and queried by users.

We have demonstrated VAPP's ability to log detailed memory operations, lock usage, actions of multiple threads, and function calls. By storing logs in a SQLite database, users can view all logged data, query the log with standard SQL queries, and implement custom low-level analysis with

the queries. The VAPP replay library facilitates loading the database and simulating program execution on the desired architecture. For parallel programs executing on multiple cores, VAPP can discover unsafe locking, lock usage, and memory buffers accessed exclusively be a single thread. We consider this project to be a success and feel that we have satisfied our 100% goals as well as a one of our 125% goals, lockset verification.

While our decision to pursue the goal of trace accessibility may ultimately prevent VAPP from scaling to operate on real-world applications due to the size of the log files and I/O costs, we nevertheless believe this area of the profiler and replayer design space was under-explored and our pursuit was worthwhile. Furthermore, by taking a log-and-replay approach instead of simulating the target architecture during application execution as is done in some related tools, VAPP is ideal for developers who intend to simulate one particular execution trace many times on different architecture configurations. Conversely, VAPP is not as well-suited for applications that run on new input with every modification to the target architecture. It will still function in such cases, but the overhead may make it a less attractive than alternatives that perform simulation alongside application execution.

One limitation of VAPP is inherent in dynamic analysis tools for correctness in parallel programs and as such cannot be easily overcome. That is, a dynamic tool can only detect concurrency bugs that occur during program execution and cannot guarantee that failure to detect such bugs means they will not arise under different conditions. However, it has been shown that given a particular execution of a program that uses multiple semaphores, detecting races or determining all possible orderings that could lead to an indistinguishable execution is NP-hard [Netzer and Miller 1990]. As such, we certainly do not consider this limitation to be due to a weakness of VAPP.

Now the we have implemented the core VAPP framework and basic replay and analysis functionality, there are many directions for possible future work. If VAPP is to become competitive with related profiling and simulation tools, the size of the log files when tracing medium to large scale programs must be addressed. Our control library to dynamically disable logging during certain parts of execution is a step in the right direction, but does not help when the programmer wishes to store and replay a full trace. In addition, because VAPP does not take a snapshot of the current system state when the trace is turned off, it is possible that replaying only those portions of the application execution that were logged could result in simulations that differ from the original execution due to discrepancies in the initial system state. Thus, more sophisticated techniques for reducing the log size must be investigated, ideally without sacrificing interpretability of the logs. Previous approaches for reducing the amount of data that must be logged, such as Netzer's transitive reduction [Netzer 1993], can likely be introduced to VAPP or custom compression techniques could be developed. In addition, VAPP currently bases all of its analysis on a single logged execution trace, but since the logs are saved in a database it is feasible to construct more sophisticated forms of analysis that span multiple executions. This would enable programmers to study how features of the underlying architecture affects performance in ways that are dependent on the input data or control flow.

We believe that expanding VAPP to provide more rich feedback to developers of parallel applications is the most exciting prospect for future development. VAPP is already able to analyze how threads use shared memory, and a natural next step would be to feed such information back to the hardware. Set pinning in shared caches in multiprocessors as well as Non-Uniform Memory Architecture systems could benefit from such feedback by localizing data to the processors that use it exclusively. Another form of feedback that could be very useful to the compiler or hardware would be calculating lock contention statistics from the already tracked log usage patterns. This would allow the system to use simple locks for very low contention locks and more space-intensive or complex locks (e.g. queue-based or list-based locks) only as needed. However, because VAPP analysis is done offline, one would either need to assume that multithread memory usage patterns do not significantly vary from execution to execution or that trends could be generalized by running VAPP on traces from multiple executions.

It would also be very interested to reunite our two Pin tools so that users could replay parallel programs. Deterministic replay is an important area of research, and although VAPP does not presently does not track parallel execution at the same level as other (hardware-assisted) approaches [Narayanasamy et al. 2005], replaying execution based on the order in locks are acquired and released would provide some form of deterministic replay. As with the aforementioned feedback mechanisms, the data needed for such a feature is already logged and available, which suggests such an extension would indeed be feasible.

6.1 Work Distribution

Work was split equally between all groups members, therefore each of the groups members is to be credited for 50% of the work.

References

L. DeRose, K. Ekanadham, J.K. Hollingsworth, and S. Sbaraglia. SIGMA: A simulator infrastructure to guide memory analysis. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–13. IEEE Computer Society Press Los Alamitos, CA, USA, 2002.

M. Martonosi, A. Gupta, and T. Anderson. MemSpy: analyzing memory system bottlenecks in programs. In *Proceedings of the 1992 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 1–12. ACM New York, NY, USA, 1992.

Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. *SIGARCH Comput. Archit. News*, 33(2):284–295, 2005. ISSN 0163-5964. doi: http://doi.acm.org/10.1145/1080695.1069994.

R.H.B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proceedings of the 1993 ACM/ONR workshop on Parallel and distributed debugging*, pages 1–11. ACM New York, NY, USA, 1993.

R.H.B. Netzer and B.P. Miller. On the complexity of event ordering for shared-memory parallel program executions. In *Proceedings of the 1990 International Conference on Parallel Processing*, 1990.

M. Ronsse and K. De Bosschere. Non-intrusive on-the-fly data race detection using execution replay. In *Proceedings of the Fourth International Workshop on Automated Debugging*, 2000.

S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

Min Xu, Rastislav Bodik, and Mark D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. *SIGARCH Comput. Archit. News*, 31(2):122–135, 2003. ISSN 0163-5964. doi: http://doi.acm.org/10.1145/871656.859633.