End-to-End Memory Behavior Profiling with DINAMITE

Svetozar Miucin
Electrical and Computer Engineering
University of British Columbia
Vancouver, Canada
smiucin@ece.ubc.ca

Conor Brady
School of Computing Science
Simon Fraser University
Burnaby, Canada
cbrady@sfu.ca

Alexandra Fedorova
Electrical and Computer Engineering
University of British Columbia
Vancouver, Canada
sasha@ece.ubc.ca

ABSTRACT

Performance bottlenecks related to a program's memory behavior are common, yet very hard to debug. Tools that attempt to aid software engineers in diagnosing these bugs are typically designed to handle specific use cases; they do not provide information to comprehensively explore memory problems and to find solutions. Detailed traces of memory accesses would enable developers to ask various questions about the program's memory behaviour, but these traces quickly become very large even for short executions. We present DINAMITE: a toolkit for Dynamic INstrumentation and Analysis for MassIve Trace Exploration. DINAMITE instruments every memory access with highly debug information and provides a suite of extensible analysis tools to aid programmers in pinpointing memory bottlenecks.

CCS Concepts

•Software and its engineering → Software performance; *Frameworks;*

Keywords

instrumentation, memory optimization, LLVM, Spark Streaming

1. INTRODUCTION

Software engineers that tune performance of their programs face many challenges, one of the most difficult is optimizing the use of the processor memory hierarchy. It is well known that tuning data structure layout and data access patterns can significantly improve performance often by orders of magnitude [9] [8] [13].

Understanding memory behavior is a notoriously difficult task. Previously available tools either target a single memory problem, lack the source level information attributed to the accesses, or are tied to a specific platform or programming language. DINAMITE is designed to address these shortcomings and give performance-minded software engineers a flexible end-to-end toolkit for understanding memory behaviour of their programs. Our system provides users with information about every memory access, every memory allocation event and every executed function, enriched with debug information that connects data obtained at runtime with its corresponding source level constructs. We created an LLVM compiler pass that instruments the relevant events, augmented with debug information, to produce event traces during the execution. Analyzing a program with DINAMITE is not tied to a specific platform or a programming language. DINAMITE can instrument any language for which there is an LLVM front-end compiler. To date, these include C, C++, Java, D, Haskell, Objective-C, Swift, Ruby, and others.

Full access traces of a program can easily reach hundreds of gigabytes even for short executions. Storing these logs and building software to analyze them efficiently is a challenge. DINAMITE provides an analysis framework implemented with Spark Streaming [14] that lets users write their analysis tools in a few lines of Scala, relying on the streaming computation model widely used for processing data. The loosely coupled design of our system is intended for easy extensibility in case the user prefers storing their logs in a filesystem, database, or even integrating it with other logging or analysis frameworks, such as Google Dataflow [5] or Kafka [7].

2. SYSTEM DESIGN

DINAMITE consists of three main components:

- The LLVM compiler pass
- A set of shared libraries that produce execution logs
- The framework for processing and analysis of execution logs

A high-level view of the system architecture is shown in Figure 1. A typical workflow for DINAMITE begins with compiling a program using the DINAMITE LLVM compiler. The desired events in the code are instrumented with calls to externally linked functions that reside in the logging library. The next step is to choose the logging library to link with the instrumented binary. This choice depends on how the programmer wishes to analyze the execution logs. For example, if they wish to persist the logs to disk, they will chose the library implementation that writes log entries into a file. If the programmer wishes to analyse traces in real

FSE'16, November 13–18, 2016, Seattle, WA, USA
© 2016 ACM. 978-1-4503-4218-6/16/11...$15.00
http://dx.doi.org/10.1145/2950290.2983941

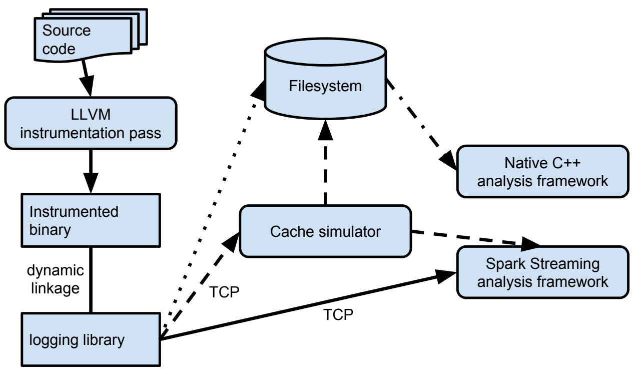

Figure 1: DINAMITE system diagram

time using Spark Streaming, they will choose the TCP output library, which streams the log records over a TCP socket to the Spark consumer.

DINAMITE provides a simple C++ framework for writing and executing analysis kernels. If the log records are streamed into the Spark Streaming engine, the final step involves using one of the existing analysis kernels, or a new custom kernel, to analyze the data. DINAMITE was designed to be modular, so it allows users to enrich the streamed log events, for example by piping them through the cache simulator (based on Dinero [6]), which will augment the memory access events with cache statistics (indicating whether this access missed in any level of the cache hierarchy), before handling it to the analysis tool.

The rest of this section describes the three parts of DINAMITE in more detail. For an even more detailed insight, we refer the reader to an extended report [11].

DINAMITE source code with its associated tools is available at https://goo.gl/7fFcVS [1].

2.1 LLVM Instrumentation Pass

DINAMITE instrumentation pass is implemented as a Module Pass in the LLVM compiler framework. Its purpose is to add logging library calls for certain events encountered in the code. The compiler recognizes and instruments the following events: *memory accesses*, *function entry/exit events* and *memory allocation function calls*. Each event is instrumented with a call to an externally linked library, which logs this event. The instrumentation pass works with the code represented in the LLVM internal representation (IR); as a result DINAMITE is able to obtain full information about the source-level constructs corresponding to the events.

Memory access events are generated each time the program accesses a memory location and are characterized by the accessed address, data type, value, the source code location of the access, variable name and thread ID. Memory allocation events represent each call to a function that allocates space on the heap. By default, DINAMITE assumes that the memory is allocated using `malloc`, but the programmer can also inform DINAMITE of any custom memory-allocation functions used in the program. The allocation events are described with the base address returned by the allocator, the size of a single allocated element, the number of allocated elements (a general case similar to `calloc`), the data type of the allocated space, the thread ID, and the source location where the allocation happened. Function events log the entry and exit for each function, and are described by the function name, the event type (entry or exit), the thread ID, and (optionally) the values of the arguments. Each string value emitted is encoded as a numeric identifier to conserve space and reduce logging overhead; these mappings are written to JSON files and are decoded at the time of the analysis.

To support different memory allocation libraries, DINAMITE loads the allocator function descriptors from an *allocator definition* file, supplied by the programmer.

2.2 Logging Libraries

The logging libraries contain different implementations of the functions called from the instrumented code during the execution. Currently, DINAMITE supports writing log records directly to the file system in text or binary format, or sending binary log records over a TCP socket.

Extending the logging library is straightforward. The programmer can customize the library using any of the twelve function prototypes. Two functions are used to setup and tear down of the logs (opening/closing a file, establishing/breaking a connection). Additional three functions are used for logging function entry and exit events, and for tracking memory allocations. The rest of the functions are used for outputting access logs for different primitive data types.

This architecture enable performance engineers to easily extend DINAMITE to produce access logs for different storage and analysis systems such as databases or different stream processing engines.

2.3 Analysis Framework

DINAMITE logs can be analyzed using native C++ or Spark Streaming tools provided in the toolkit, or the programmer can write any analysis scripts using Unix command line tools, Python or any other language of choice. Here, we focus on the Spark Streaming tools, because of its ease of use and the ability to process logs in real time.

Our analysis framework consists of two classes that extend Spark Streaming components *LogReceiver* and *LogEntryReader*. The former is an implementation of a Spark Streaming receiver that listens on a TCP socket and receives binary log data. Each log entry in the stream is stored for use within the *SparkStreamingContext*. The stored log entries are in the binary format representing a C struct. To convert them into Scala classes, they are piped through a *map* operation that invokes the extraction method of *LogEntryReader* on each entry.

From this point, the produced stream of Scala classes can be analyzed using arbitrary operations within Spark Streaming. Listing 1 shows a simple analysis kernel that counts accesses per variable in the incoming stream and outputs them in the console. Lines 1-11 show the standard setup operations that need to be present in all DINAMITE analysis tools. Lines 13-19 are the main body of the analysis tool.

2.3.1 *Cache Simulator*

To support detailed analysis of cache behavior, DINAMITE can utilize a cache simulator program, as an intermediate step between log output and analysis frameworks. We provide our implementation of a simple single level cache simulator that annotates DINAMITE logs it receives, and passes them on to the next component in the pipeline. With this setup, it is possible to attribute cache hit or miss information to each access in the stream, and use them to better understand the implications of source level constructs on cache performance.

Listing 1: Example Spark Streaming kernel

```
def main(args: Array[String]) {
  val sparkConf = new SparkConf()
    .setAppName("AccessCounter");
  val ssc = new StreamingContext(
      sparkConf,
    new Duration(1000));
  ssc.checkpoint("/checkpoints/");

  val logs = ssc
    .receiverStream(new
        LogReceiver(9999))
    .map(rawlog =>
    LogEntryReader.extractEntry(
        rawlog));

  val counts = logs
    .filter(log =>
      log.isInstanceOf[AccessLog])
    .map(access =>
    (access.as(...)[AccessLog].
        varId, 1L))
    .reduceByKey(_+_)
    .updateStateByKey(sumUpdater);

  counts.print();

  ssc.start();
  ssc.awaitTermination();
}
```

2.4 Performance

To give a better idea of where DINAMITE stands when compared to previously available instrumentation frameworks, we provide slowdown numbers (compared to uninstrumented execution) in Table 1. Intel's Pin comes with a comparable access tracing tool *Pinatrace.* Pinatrace only outputs information about the address, size of access, and type of access, which is a significantly reduced subset of DINAMITE's log information. Valgrind's MemCheck slowdown is obtained from Nethercote et al.[12]. The comparison is not fair because MemCheck only verifies validity of memory locations at runtime, and only outputs data at the execution's end. To the best of our knowledge there are no available Valgrind tools comparable to DINAMITE and Pinatrace.

DINAMITE's instrumentation and logging performance is around 10x faster than the comparable Pinatrace tool. Even when compared to MemCheck, which doesn't perform any output at runtime, DINAMITE's full instrumentation is only 60% slower. Finally, even with the full analysis pipeline running, DINAMITE's performance is comparable to performing only access trace output with Pinatrace.

Table 1: Instrumentation overhead comparison

Framework	Slowdown
Pin (pinatrace output to RAM disk)	354x
Valgrind (MemCheck)	22x
DINAMITE (output to RAM disk)	36x
DINAMITE (Spark analysis)	537x

3. CASE STUDY - SHARED VARIABLES

In multi-threaded software, it is very important to organize memory accesses in a way that avoids heavy sharing of variables between threads. Even without explicit synchronization, sharing a variable between threads can severely impact performance because of the cache coherency protocol overhead.

Previous tools that address variable sharing issues (Intel's VTune [10], DProf [13], MemProf [8]) are hardware-specific which makes them non-trivial to set up (in the case of DProf and MemProf, kernel modifications are required). DINAMITE, with its analysis framework, provides an easy to use platform for writing tools that pinpoint shared variable problem causes with precision.

We demonstrate the utility of DINAMITE on a case study involving debugging a known scalability bottleneck in WiredTiger [2] [4], a MongoDB [3] storage engine. In order to test both the precision and the ease of creating tools in DINAMITE, the student tasked with writing a shared variable detection tool was *not* told about the root cause of the problem. They were only told that a shared variable bottleneck exists, and instructed on how to replicate the problematic workload.

The engineer who originally solved the scalability issue did so in about a week, using MemProf, which required kernel modifications and communication with the authors. Applying modifications to the kernel is likely to be considered "beyond the call of duty" for many engineers. The creation of a DINAMITE analysis tool for this purpose and finding the root cause of the scalability bug took a couple of hours, done by a person familiar with the overall workings of the framework.

The analysis of WiredTiger's shared variable bottleneck was done in an exploratory fashion, with two different Spark Streaming kernels. The first kernel was used to find the most shared variables, and the second was used to refine the search and produce source locations where the majority of sharing occurred. The output of the tools was processed with simple Python scripts to produce human readable summaries.

In the first pass of the analysis, logs are mapped into tuples containing the accessed address, variable identifier and a thread identifier. These tuples are used as a key in a sum reduce operation. The result of the reduce operation represents how many times each address was accessed by each thread, which is stored persistently in the filesystem. A Python script is used to transform this data into a summary of the number of threads sharing each address, as well as the access counts and variable identifier. The data is filtered to eliminate entries based on the following criteria:

- Remove all entries accessed by a single thread
- Remove all entries don't exhibit uniform sharing, defined as:

  Let A_{sorted} be a list of all the per-thread access counts for the address, in descending order, zero indexed.

  We define all addresses for which $A_{sorted}[0] < 2 * A_{sorted}[1]$ as uniformly shared.

Table 2 shows the top 5 entries from the first pass of the analysis, output by our Python script. From this information, we can conclude that the `v` member of `__wt_stats` data

Table 2: Most accessed shared variables in WiredTiger

Address	# Accesses	# Threads	Variable
0x64D900	*42495568*	*32*	*__wt_stats.v*
0x64D1A4	26183326	74	__wt_connection_impl
0x64E0EC	7233836	72	__wt_connection_impl.N/A
0x64D100	4786616	36	__wt_txn_global.states
0x64D540	4786370	34	__wt_stats.v

structure is the most accessed shared variable, shared among 32 threads.

The second DINAMITE analysis tool is similar to the first one. It performs counting in the same fashion, except logs are now filtered to include only accesses to `__wt_stats.v` and the grouping key is enhanced with the source location information. An additional Python script is used to group the output from Spark Streaming in a human-readable format.

Listing 2 shows the final output of our analysis, identifying the shared variable bottleneck to be `__wt_stats.v` accessed on line 446 of `bt_curnext.c`. The output of the tool is a JSON document, containing the number of threads accessing the variable, the total and per-thread access counts, and the source level information about the variable. Upon consulting WiredTiger engineers, we verified that this is the correct root cause of the scalability bottleneck. Figure 2 shows the impact of this bug on the scalability of WiredTiger. The performance improvement after removing the bug reaches the factor of 20 for 32-thread executions.

Listing 2: WiredTiger shared variable analysis result (JSON)

```
{
  "threadcount": 18,
  "totalcount": 311881,
  "threads": [
   [
    156,
    19492
   ],
   [
    163,
    19520
   ],
   ...
   ],
  "file": "wiredtiger/build_posix
      /../src/btree/bt_curnext.c",
  "line": 446,
  "variable": "__wt_stats.v"
 }
```

Finally, to illustrate the amount of effort required for writing the second shared variable analysis kernel, we show the main part of its code in Listing 3. The rest of the kernel is identical to the kernel shown in Listing 1, and is omitted for brevity.

Our case study shows the utility of DINAMITE as a memory performance debugging tool. The programming interface of Spark Streaming makes it easy to write powerful analysis tools, and its engine enables easy scaling both in terms of storage and computation. Our previous work shows two more case studies which are omitted here for brevity, as well as a more detailed account of the shared variable case study presented here [11].

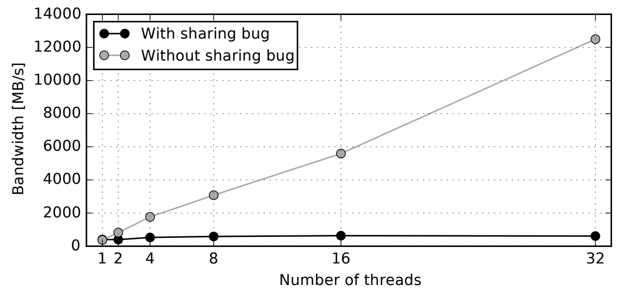

Figure 2: Scaling improvements in WiredTiger after removing the shared variable bug

Listing 3: Excerpt from the Shared Variable Analysis kernel

```
val accesses = logs
 .filter(l =>
  l.isInstanceOf[AccessLogEntry])
 .map(l =>
  l.asInstanceOf[AccessLogEntry]);

val filt = accesses
  .filter(l =>
   l.varId == 217);
// 217 is the id of __wt_stats.v

val counts = filt.map(a =>
  ((a.ptr,
    a.thread_id,
    a.varId,
    a.file,
    a.line), 1L))
  .reduceByKey(_+_);

val runningCounts = counts
    .updateStateByKey(sumUpdater);
```

4. CONCLUSION

We presented DINAMITE, a set of tools for instrumentation and analysis of memory accesses in software. DINAMITE expands on the previously available instrumentation frameworks by complementing the access traces with rich source level information which is often paramount for fully understanding memory behavior. The full pipeline of tools available in DINAMITE allows for analysis of allocation, access and cache event patterns, both offline and online using modern stream processing engines.

Our case study emphasizes the importance of good memory behavior analysis tools in modern industry workloads. DINAMITE allowed for faster debugging of an important scalability problem, without the need for understanding the target program's code base in detail.

In the future, we plan to expand on the library of readily available analysis tools in DINAMITE, in order to make memory performance debugging more accessible to developers. We also intend to further improve DINAMITE's performance, as described in our previous work.

5. REFERENCES

[1] Dinamite bitbucket project, 2016.
[2] Wired tiger: making big data roar, 2016.

[3] Wiredtiger storage engine, 2016.

[4] Wt-2029 improve scalability of statistics, 2016.

[5] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, 2015.

[6] J. Edler and M. D. Hill. Dinero iv trace-driven uniprocessor cache simulator, 1998.

[7] J. Kreps, N. Narkhede, J. Rao, et al. Kafka: A distributed messaging system for log processing. NetDB, 2011.

[8] R. Lachaize, B. Lepers, and V. Quéma. Memprof: A memory profiler for numa multicore systems. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*.

[9] C. Lattner and V. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *ACM SIGPLAN Notices*, volume 40.

[10] R. K. Malladi. Using intel® vtune^{TM} performance analyzer events/ratios & optimizing applications. *http:/software. intel. com*, 2009.

[11] S. Miucin, C. Brady, and A. Fedorova. DINAMITE: A modern approach to memory performance profiling. Technical report, 2016.

[12] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42.

[13] A. Pesterev, N. Zeldovich, and R. T. Morris. Locating cache performance bottlenecks using data profiling. In *Proceedings of the 5th European conference on Computer systems*.

[14] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Presented as part of the*, 2012.