Gleipnir: A memory tracing and profiling tool

Tomislav Janjusic, Krishna Kavi

University of North Texas, Denton Texas

Abstract

Embedded and high performance applications often require fine-tuning to improve their performance. This is achieved using analysis tools that provide insights into the application’s behavior. A common approach is to instrument the application code and observe its behavior during an actual execution on a target system.

In this paper we describe a program profiling and tracing tool called Gleipnir. Gleipnir is built as a plug-in tool to a widely used binary instrumentation framework called Valgrind. Gleipnir can be used to trace memory accesses and associate each access with a specific program internal structure such as threads, functions, local, global, and dynamic data structures, and scalar variables. This ability makes Gleipnir a good candidate for advanced memory performance tuning.

The data provided by Gleipnir may be used by trace-driven simulators, such as cache simulators to analyze accesses to data structure elements so that programmers can understand how the memory access patterns are impacting the execution time of the application. The programmer may then be able to change data layouts or reorder code to change the access patterns and eliminate performance bottlenecks.

The goal of Gleipnir is to give information rich traces that can be used by any number of advanced memory analysis tools, particularly cache simulators. Our claim is that despite advances in allocation techniques and data reordering, detailed dynamic and static memory behavior of applications is often not readily available or available only in terms of statistical average accesses and cache miss rates.

It is our hypothesis that optimizing cache performance at all levels is very important to improving the performance of applications running on single-core and multi-core processors. In this paper we will describe Gleipnir and provide examples of how the output of Gleipnir can be used by cache simulation tools.
simulators to understand the impact different data organization on memory access latencies. The overall goal of our research is to develop techniques that can be used by application developers as well as compilers and runtime systems to improve performance of applications running on single and multi-core processors.

Keywords: Binary instrumentation, Memory Profiling, Cache Simulation

1. Introduction

Processor-Memory speed gap is still a major hurdle to performance. Software developers need to analyze application's memory access behavior to maximize the execution performance. For this purpose software developers often rely on instrumentation and profiling tools. These tools are used in combination with other performance optimization, tuning, and analysis software. For example, today's processors are shipped with hardware performance counters. There are architecture specific counters that count various hardware related events. For example cache miss counters may count the number of cache misses occurring during a program execution. The programmer can access the hardware counters directly from the application by setting specific register values and executing a system interrupt. The data gathered by the counters is collected and analyzed offline. Because the process of setting up hardware counters is cumbersome, designers developed tools, e.g. performance counter libraries such as the performance application programming interface (PAPI)[1], that provide application programing interfaces (APIs) for easier hardware counter set-ups.

Broadly speaking, application profiling tools can be categorized based on their data gathering methodologies and functional scope. There are performance libraries, event based and sampling, binary instrumenting, and debugging tools. Instrumenting tools are further categorized into compiler assisted, binary translation, binary instrumentation, and hybrids or runtime code manipulation tools. Binary instrumentation tools are further categorized into static and dynamic. Common dynamic binary instrumentation and runtime code manipulation frameworks cited are Pin Tool[2], DynInst[3], DynamoRIO[4], and Valgrind[5]. These frameworks come with a vast number of plug-in tools. Virtually every framework comes with a basic tracing tool and basic cache simulators.

To differentiate the different terminologies in this paper we will refer to
the profiling tool, or simply tool, as the instrumented code inserted by the framework. We refer to the framework as the underlying mechanism, or core-tool, that enables the profiling tool to insert instrumented code. Finally the client is the application that is instrumented.

1.1. Fine grained memory access analysis

There are many frameworks that enable the development of memory access analysis tools. Plug-in tools are limited by the ability of the framework to expose application related information. For example, exposing address traces limits the tools in performing more advanced analysis or tracing. When using basic traces, i.e. access type, address, and size, a cache simulator is limited to simulate only basic information such as overall behavior, number of memory read and writes due to cache misses, number of capacity, compulsory, and conflict misses, etc. For advanced cache analysis we need more advanced tools that offer greater detail about an application’s cache behavior. Valgrind’s framework comes with tools such as Cachegrind[6] and Callgrind[7] that analyze an application’s cache behavior. These tools can classify the collected data per source code line, function, etc. They fail however, to relate the source of the cache misses back to the root cause, such as which data items are conflicting and in which order.

Similarly, Dprof[8] is a tool that identifies cache critical data structures by tracking instructions that resulted in most misses. Dprof uses hardware performance counters and correlates this information by translating instructions that resulted in a cache miss back to source code. This is done by tracing the instruction address and using a debug parser. However, the tool does not identify conflicting data structures.

In order to understand and simulate this behavior we must correlate a data access to its source code level data structure name, and we must keep track of all evictions that persist throughout a program’s runtime. Moreover, to enable visual representations we must also keep track of specific cache sets and corresponding subsets. Providing this information can help programmers and application developers visualize conflicts of program data-structures. Therefore, when we set out to develop Gleipnir [9] we aimed at exposing as much information as possible to cache and memory analyzers. We achieved this using a heavy-weight binary instrumentation framework that comes with the necessary infrastructure to enable fine grained memory tracing. We opted for Valgrind predominately because of the ability to easily
track low level debug information and the availability of source code which can be modified for our purpose.

Our goal is to provide detailed tracing information for every memory access and relate the access back to a source code data elements. In our earlier Gleipnir version [9] we were able to trace accesses and relate them to static and global variables. The current version can trace accesses to static, global, and dynamically allocated objects and relate them back to source-code variable and structure names. We have also greatly enhanced our simulation environment to trace fine-grained cache usage. Using Gleipnir’s traces cache simulators can analyze all data structure related cache misses and relate misses to their root cause. Our approach is different from existing tools because Gleipnir itself does not offer trace analysis, but instead offers information rich enough with traces for use by analysis tools. To illustrate how Gleipnir traces can be used, we modified Dinero, a trace-driven cache simulator. In this paper we outline some analyses that can be performed using the traces generated by our Gleipnir tool. However we feel that many more analyses can be developed using the very rich information provided by Gleipnir, and it is fairly easy to modify Gleipnir to generate additional information with traces.

The rest of the document is organized as follows; in Section 2 we review related work. In section 3 we will discuss Gleipnir and its tracing capabilities. Our analysis environment, i.e. our cache simulator, is discussed in Section 4 by following an example run and looking at various cache information provided by the simulator. In section 5 we will discuss our research’s future direction. Section 6 finalizes with our conclusions.

2. Related Work

Application tuning and optimization is generally performed using profiling tools, performance counters, or a combination to deliver hardware and application related metrics. The goal of gathering performance and profiling data methodologies is two fold. Firstly, the information has to be detailed enough to help the programmer understand where the bottlenecks are, and secondly the information must be detailed enough so that optimizations can be applied. Profiling tools may consist of any combination of three classic approaches: hardware counters, simulations, and analysis models.

• Hardware counters are special purpose registers available with modern processors that enable sampling and counting of hardware events
as they occur during program execution. Typical events that can be sampled include cache hits and misses, bus transactions, hardware interrupts. Hardware counters are processor specific and any information gathered is applicable to a specific run, meaning that one needs to re-execute the application and collect data after each modification or optimization.

An example tool that utilizes hardware performance counters for cache analysis is DProf[8]. DProf identifies cache critical data-structures, i.e. structures that are most detrimental to cache performance. By combining an instruction based sampling approach, utilizing performance counters, and parsing debug information DProf can track structures that have incurred the most hits and misses. The sampling rate determines the accuracy of collected data and impacts profiling execution time. Since DProf uses hardware counters, any changes to application code requires re-execution using DProf; and each execution may result in slightly different cache miss information because of sampling, and may potentially result in misidentification of structures as the ones causing most misses. In constrast, Gleipnir does not use hardware counters but produces traces with very detailed information. DProf does not account for system libraries and functions that may be the root cause of cache misses encountered by an application. Other tools that utilize hardware performance counters for various profiling various timing and memory related effects are TAU[10], OpenSpeedShop[11], HPCToolkit[12], etc.

- Profiling using simulators is another approach to measure application’s cache behavior. The advantage of using simulators instead of actual hardware is that it allows for full control over hardware components such as the CPU, CPU’s cache or other memory components. The drawback of using simulations is the excessively long simulation times, and the potential loss of accuracy when compared to hardware counters, since it is very difficult to simulate complex multi-issue, out-of-order and speculative instruction execution. Simulators such as Simics[13], SimpleScalar[14], DineroIV[15] provide modules for cache design space exploration, but more importantly they can be, and often are, used to evaluate an application’s memory behavior.

- Analysis models allow static analysis of application source code or bi-
nary images. Because they are static the cost models can be encoded directly in the compiler. The goal of these models is to identify memory access and use the cost to either change data layouts or change code to modify access patterns. We must note that these techniques may be poor at predicting runtime events observable only at runtime, e.g. branches, cache evictions and references, structure allocations etc.

To observe an application’s memory behavior researchers must be able to gather application’s memory access patterns and map the access to multiple levels of memory subsystems. Collecting a memory access pattern can be achieved by executing the application and collecting traces or by analyzing an application’s binary image statically. Once traces are collected, they can be analyzed by discrete-event simulator such as a trace-driven cache simulators or just-in-time plug-in tools that simulate cache behavior during profiling.

There are several frameworks that enable the development of just-in-time analysis, including Valgrind[5], Pin[2], DynInst[3], and DynamoRIO[4].

Examples of actual plug-in tools for memory analysis include CacheGrind[6], Callgrind[7], Cache Pintool, CMP$im[16], SIGMA[17], and others.

CacheGrind, a Valgrind based cache simulator, summarizes cache statistics and relates them to source code lines. This allows the programmer to identify potential bottlenecks at source code level. Callgrind[7] is a callgraph profiler built on top of CacheGrind. Callgrind provides cache hit and miss statistics for each function of a program. This information is useful in identifying poorly performing functions in terms of cache misses. Callgrind is functionally similar to CacheGrind; it tracks memory events and utilizes a built-in cache simulator to gather cache statistics. PinTool comes with a simple cache simulator that provides summary cache statistics. CMP$im is a more advanced Pin cache simulation tool designed for cache-design space exploration. CMP$im can simulate a multi-core environment. Sigma is a DynInst based tool that identifies memory bottlenecks related to inefficient data layouts.

Sigma is similar to Gleipnir in data representation as it provides misses for data structures but it does not correlate cache conflicts among various structures.

In order to compare and contrast other tools in this area we must take note of what Gleipnir is and what it is not. Gleipnir is a memory tracing tool; the analysis of the traces is external to Gleipnir. For this paper we use a modified version of DineroIV[15] cache simulator. Decoupling the trace
collection from the simulation offers several benefits. In addition to cache analysis other analyses of traces are possible. Gleipnir’s traces provide very rich information on every application’s memory access. This information may be used for a variety of research areas as well as performance profiling.

3. Implementation Overview

3.1. Valgrind’s Intermediate Representation

Valgrind’s framework consists of a core-tool and one or more instrumentation tools. The core-tool operates on sections of code blocks (or SuperBlocks), a collection of up to 50 instructions with a single entry and multiple exits. SuperBlocks are disassembled and converted to an intermediate representation (IR). The IR is used by the instrumentation tools for inserting code to collect relevant information. The core-tool recompiles (or resynthesizes) the instrumented IR block. The recompiled code with instrumentation is then executed on a simulated processor. The execution of the application and the instrumentation code will provide desired information about the application. Figure [1] shows a diagram describing this process.

![SuperBlock flow chart](image)

Figure 1: SuperBlock flow chart

Valgrind and client application operate from the same user space. This means that every instruction is simulated on a synthetic CPU. There is an inherent slowdown (of at least 4 times) because Valgrind transforms and executes the transformed instruction stream on a simulated CPU. Figure 2 shows the difference of executing applications (we refer to them as client applications because we collect data on these applications) with and without Valgrind.

3.2. Tracing Instructions

Valgrind is supplied with some tools which can be used to trace the executed instruction stream. Gleipnir instruments instruction events identified
by Valgrind’s intermediate representation. It also utilizes debug parsing and wrappers on dynamic memory allocation functions to identify source level structures with traces. Each event in the instrumented code is either an Instruction read (Ir), Data read (Dr), Data write (Dw), or Data modify (Dm). When the instrumented code is executed an instruction or data address is passed to Valgrind’s debug parser. The debug parser searches the symbol table and returns the information about the source level structures back to Gleipnir. We modified Valgrind’s debug parser functions for our purpose.

Symbol table look-up enables Gleipnir to deliver fine grained debug information for each data write, read, or modify. Table 1 and 2 show the format of a typical line of trace produced by Gleipnir.

<table>
<thead>
<tr>
<th>Access Type</th>
<th>Address</th>
<th>Size</th>
<th>Segment</th>
<th>Thread Id</th>
<th>Function</th>
<th>Scope</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>S</td>
<td>M Ins</td>
<td>vAddr</td>
<td>size</td>
<td>S</td>
<td>G</td>
<td>H</td>
</tr>
</tbody>
</table>

Table 1: Gleipnir’s basic trace line.

<table>
<thead>
<tr>
<th>Misc</th>
<th>Ret. Pointer</th>
<th>Thread Id</th>
<th>KEYWORD</th>
<th>Size</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Ret. Addr</td>
<td>tid</td>
<td>MALLOC</td>
<td>size</td>
<td>Name</td>
</tr>
<tr>
<td>X</td>
<td>N/A</td>
<td>tid</td>
<td>CALLOC</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>X</td>
<td>N/A</td>
<td>tid</td>
<td>REALLOC</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>X</td>
<td>N/A</td>
<td>tid</td>
<td>System Call</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 2: Special keywords and other simulator related information.

The first field is the access type, either a Load, Store, Modify, or X (for miscellaneous instructions). The second field is the virtual address\(^1\) of the

\(^1\)Gleipnir is also capable of producing physical addresses as shown in section 5 for shared memory analysis.
data being accessed followed by its size.

Next several fields relate the access to segment, stack, global, or heap/dynamic, and the thread that executed the access. The extracted debug information supplements the trace line with the function name that contained the access. If any symbol information exists the trace line will be annotated with the element’s scope, Local or Global, and the element’s type variable or structure. If the access is to a dynamically allocated structure then the scope will be heap which is followed by the allocated number (indicating multiple allocation of the structure during the program execution). The final field in the trace is the name of the variable. If the accessed element is part of a structure then both the structure name and the element within the structure are shown. Likewise if the structure name contains multiple nestings then Gleipnir identifies that as well. Table 2 shows a special trace line intended for simulation or analysis purposes. For special keywords such as when an allocation occurs, threads spawns or joins, or a client executed a fork and other system calls, Gleipnir will output that information into a special trace line annotated by an X access type. Gleipnir relies on Valgrind’s internal debug parser; therefore, applications for which the simulator needs local and global structure/variable information must be compiled with the compiler’s -g flag.

3.3. Tracing static, global, and dynamic elements

Because local and global elements are known at compile time the compiler will use symbol tables to store information about the variables. Debug information is not available for dynamically allocated structures and we instrument allocation functions to track each allocation, trace the source code that invoked the allocation, and extract the name of the structure involved in the allocation (if the name is available).

Listing 1 shows an example source code which references local and global structures. Listing 2 shows a portion of the trace generated by Gleipnir for the source code of Listing 1, relating the source level information with memory accesses. This information can be useful for analysis tools that aim to improve performance by observing the accesses to different program variables.

The source code in Listing 1 and 2 include only static and global variables (and no dynamically allocated structures). This example is included so that the reader can observe the level of detail provided by Gleipnir. The source code’s main function starts with a Gleipnir specific macro that turns the
instrumentation on. With the macros GLEIPNIR_START_INSTRUMENTATION and GLEIPNIR_STOP_INSTRUMENTATION the user can instrument only the part of the source code that is of interest.

The code in our first example includes global and local structures and variables. The function functionA is defined in lines 8-15. In lines 17-30 the main function declares some structures and scalar elements. In lines 22-28 the structures and scalar elements are accessed. Finally in line 21 function functionA is called and the corresponding code in lines 8-16 are executed.

Listing 1: Example 1 source code

```c
int my_gl, my_glArray[10];
struct type {
    int A;
    int B[10];
};
struct type GStruct[2];

void functionA ( int param1 ) {
    int fnc_local;
    param1 = 12345;
    fnc_local = 1234;
    my_gl = 543;
    my_glArray[2] = 123;
}

int main ( void ) {
    GLEIPNIR_START_INSTRUMENTATION;
    int main_local = 100;
    functionA ( main_local );
    my_gl = 234;
    my_glArray[2] = 123;
    GStruct[1]. A = 5;
    GStruct[0]. B[3] = 123;
    GLEIPNIR_STOP_INSTRUMENTATION;
    return 0;
}
```

Listing 2: Example 1 trace

```
START PID 27922

S 7ff000160 8 1 S main
S 7ff00016c 4 1 S main LV main_local
L 7ff00016c 4 1 S main LV main_local
S 7ff000120 8 1 S main
S 7ff000118 8 1 S functionA
S 7ff000110 4 1 S functionA
S 7ff000114 4 1 S functionA LV fnc_local
L 7ff000114 4 1 S functionA LV fnc_local
S 000601098 4 1 G functionA GV my_gl
S 0006010a8 4 1 G functionA GS my_glArray[2]
L 7ff0001120 8 1 S functionA
S 000601098 4 1 G main GV my_gl
S 0006010a8 4 1 G main GS my_glArray[2]
S 00060106c 4 1 G main GS GStruct[1]. A
S 000601050 4 1 G main GS GStruct[0]. B[3]
S 7ff000130 8 1 S main LV _zzq_args[0]
S 7ff000128 8 1 S main LV _zzq_args[1]
S 7ff000140 8 1 S main LV _zzq_args[2]
S 7ff000148 8 1 S main LV _zzq_args[3]
S 7ff000150 8 1 S main LV _zzq_args[4]
S 7ff000158 8 1 S main LV _zzq_args[5]
L 7ff00012c 4 1 S main
L 7ff00012c 4 1 S main
```

The program’s execution is observable from the corresponding trace in Listing 2. Each trace line represents the data elements accessed during the program execution. The trace starts with the main function storing a value to the local variable main_local, trace line 3-4, corresponding to the source code line 20 on the left.

The function functionA is traced in lines 6-13. The trace lines show the store to the functions parameter param1 followed by several local and global stores. At each line the trace identifies the segment of the data (Stack, Global, or Heap), thread id (useful when multiple threads are involved), the function that caused the access, the scope of the variable accessed (LV, LS, GV, or GS), and the name of the variable, structure, or structure element that is accessed, including the array index when arrays are accessed.
Because debug information is not available for dynamically allocated objects, Gleipnir captures the necessary information using wrappers around allocation (i.e. malloc) functions. When an allocation routine is executed Gleipnir will record the allocated structure’s size and the base address. Gleipnir will use the source line that contained the call to the allocation function to extract the name of the structure (if available), and also identify the instance of the allocation when the same structure is involved in multiple allocations. When an access to a dynamically allocated region is encountered, the address of the access is used to locate the name of the structure, the instance of the allocation and name of the element that is involved. A dynamic trace line is different from a static or global trace line in terms of the related debug information. Dynamic debug information is replaced by the structure name if applicable and the number of offset bytes into the structure.

Listing 3 is an example of a simple program that allocates a structure and accesses the elements within that structure. Listing 4 shows the corre-
sponding trace when the program in Listing 3 is executed\textsuperscript{2}.

The code defines simple structures with several elements in lines 1-12. The structure is allocated in lines 18-19 and lines 24-25. The pointer \textit{ptr} is used to point to the allocated structure. The code accesses the elements via \textit{ptr} in lines 20-22 and lines 26-28.

Listing 4 shows the resulting trace. Notice that in this trace we can also observe the memory accesses generated by the \textit{malloc} routine. If the debug information is available Gleipnir can map accesses by libraries as well. However, for libraries without debug information the trace will map up to function level granularity.

The trace in Listing 4 shows the executed code. The first instruction is a special keyword that indicated an allocation. The keyword X is used in our cache simulator as any non-executable instruction and ignored in computing cache statistics. The special keyword shows the return address of the allocated structure, size (24 allocated bytes), and the name of the allocated structure. Next several instructions are malloc library instructions. Trace lines 8-19 correspond to source code lines 18-22. In the trace the pointer value is loaded in line 10 and a store to element \textit{valInt} is traced in line 11. Notice that the store to element \textit{valInt} is an access to a dynamic memory region; therefore, the trace line indicates a scope level \textit{H} for heap and enumerated as 0 because this the first instance of allocated block. It also includes the structure name and the offset of the allocation.

Gleipnir’s traces allow memory analyzers and cache simulators to report cache behavior which correlate misses between local, global, and dynamically allocated regions.

3.4. Multi-threading

Gleipnir is a Valgrind plug-in instrumentation tool and our tracing ability is constrained by the limitations of Valgrind. Multithreading in Valgrind is fully supported and Valgrind can be used to trace POSIX and OpenMP threads. However, Valgrind is not a multithreaded platform. This implies that thread execution under Valgrind is serialized. Scheduling of threads can be left to the operating system or flags can be passed to Valgrind to enforce fair scheduling mechanisms. Tracing multithreaded applications will result in traces for each thread for each scheduling slice and the traces of different

\textsuperscript{2}We have omitted some trace lines for clarity purposes.
threads appear interleaved within the scheduling slice. To obtain a more realistic image of memory accesses for multithreaded applications, analysis tools will have to rely on appropriate models for interleaving accesses from different threads (e.g., interleaving on every execution cycle).

Listing 5: Example source code

```c
int main(int argc, char *argv[1]) {
    GLEIPNIR_START_INSTRUMENTATION;
    int i=0, rc;
    long t=0;
    pthread_t thread_ids[NUM_THREADS];
    /* create threads */
    for (t =0; t < NUM_THREADS ; t ++){
        rc = pthread_create(& thread_ids[t],
                NULL , thread ,
                        ( void *) t);
    }
    while (lock >0){
        pthread_mutex_unlock(& mutex);
        pthread_mutex_lock(& mutex);
    }
    GLEIPNIR_STOP_INSTRUMENTATION;
    pthread_exit( NULL);
    return ;
}
```

Listing 6: trace-file snippet

```c
#include <pthread.h>
#define NUM_THREADS 5
int lock = NUM_THREADS;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
void sumnumber(int sum){
    int i;
    int _sum=0;
    for(i=0;i<sum;i++){
        _sum+=i;
    }
    return ;
}
void * thread ( void * threadid )
{
    long tid = ( long ) threadid ;
    int i;
    sumnumber(tid +5);
    pthread_mutex_lock (& mutex);
    lock--;
    /* release lock */
    pthread_mutex_unlock (& mutex);
    pthread_exit ( NULL);
}
```

The code in Listings 5 and 6 show an example program that uses multiple threads. In lines 9-14 Listing 5 shows the program’s `main` function that creates several threads within the `for` loop. When all threads are created the main thread waits for a lock to be released (see lines 16-19). Lines 9-30 in Listing 6 show the threads calling a `sumnumber` function that adds numbers. When finished the thread will execute a `lock_mutex` and `release_mutex`.

Listings 7 and 8 show Gleipnir generated traces for the code in Listings 5 and 6. In Listing 7 the main thread calls the system function which creates the threads. When a thread is created the main thread will insert a miscellaneous instruction (in line 7) `X THREAD_CREATE 1:2`. This line indicates that thread 2 was created by thread 1. These instructions aid in modeling a multithreaded cache behavior. Depending on the thread mechanism the newly created threads will compete over the CPU time slice. In our example we have enabled a fair scheduling mechanism that enqueues newly created threads in an execution queue. We can observer the thread execution in trace lines 9-32. Listing 8 shows the execution of subsequent threads.
We only show relevant trace lines and omit other trace lines to simplify the listings.

### Listing 7: Example source code

```c
L 7ff0001e0 8 1 G main LV tid 
S 7ff000178 8 1 S main 
S 7ff000170 8 1 S 
S 005c09ef0 8 1 G clone 
X THREAD_CREATE 1:2 
S 005c09ef0 8 2 G start_thread 
S 005c09e8 8 2 G 
L 005c0ad40 8 2 S start_thread 
S 005c09ee8 8 2 S start_thread 
S 005c09ee0 8 2 G thread 
S 005c09ec8 8 2 G thread LV threadid 
L 005c09ec8 8 2 G thread LV threadid 
S 005c09ed8 8 2 G thread LV tid 
L 005c09ed8 8 2 G thread LV tid 
S 005c09eb8 8 2 S thread 
S 005c09eb0 8 2 S sumnumber 
S 005c09e9c 4 2 G sumnumber LV sum 
S 005c09eac 4 2 G sumnumber LV _sum 
L 005c09eb8 8 2 S sumnumber 
L 000601020 8 2 G 
L 000601070 4 2 G pthread_mutex_lock GV mutex 
L 000601060 4 2 G pthread_mutex_lock GV mutex 
L 00680aeb8 8 3 S sumnumber 
S 00680ae0c 8 3 S thread LV sum 
S 00680ae8c 8 3 S thread LV _sum 
S 00680ae8e 8 3 S thread LV tid 
M 00060106c 4 2 G pthread_mutex_lock GV mutex 
S 0006010a8 8 3 S pthread_exit 
L 0006010d8 4 2 G pthread_mutex_lock GV mutex 
```

### Listing 8: trace-file snippet

```c
L 7ff0001e0 8 1 G main LV tid 
S 7ff000178 8 1 S main 
S 7ff000170 8 1 S 
S 005c09ef0 8 1 G clone 
X THREAD_CREATE 1:2 
S 005c09ef0 8 2 G start_thread 
S 005c09e8 8 2 G 
L 005c0ad40 8 2 S start_thread 
S 005c09ee8 8 2 S start_thread 
S 005c09ee0 8 2 G thread 
S 005c09ec8 8 2 G thread LV threadid 
L 005c09ec8 8 2 G thread LV threadid 
S 005c09ed8 8 2 G thread LV tid 
L 005c09ed8 8 2 G thread LV tid 
S 005c09eb8 8 2 S thread 
S 005c09eb0 8 2 S sumnumber 
S 005c09e9c 4 2 G sumnumber LV sum 
S 005c09eac 4 2 G sumnumber LV _sum 
L 005c09eb8 8 2 S sumnumber
```

3.5. Multiprocess capabilities

Multiprocess capabilities are also fully supported. Applications that create multiple processes using system calls, e.g. `fork()`, can be traced by Gleipnir (to the extent permitted by Valgrind). Valgrind will fork its own instrumenting image so that each process can be separately instrumented. Gleipnir includes a mechanisms to detect when a child process is created. This will result in a new trace being created for each child process along with traces for the parent process. The parent trace will be annotated with a keyword `X FORK PID`, where PID is the process ID. This allows cache simulators to model multi-process simulations. Figure 3 illustrates this scenario.

4. Analysis Environment

The traces produced by Gleipnir can be used by other analysis tools. In our research we are interested in the cache memory performance and we use a modified version of Dinero IV[15]. Our modifications are primarily to force DineroIV to accept Gleipnir’s traces thereby extending its simulation
Figure 3: Forking a child process of a client.

capabilities. In this section we will illustrate our analyses: because Gleipnir provides very fine-grained information on accesses, we can analyze the cache hits and misses per function or even per program variable and understand cache conflicts among variables. Such analyses can then be used to change data layouts, change the disposition of elements of a structure or refactor code to change access patterns.

4.1. Analysis Cycle

A typical analysis procedure involves three steps as outlined in Figure 4. A user runs the application through Gleipnir. This generates the trace file which can be analyzed by appropriate analysis tools. At present we provide the modified Dinero IV along with Gleipnir. Other analysis tools can be used with Gleipnir, provided the analysis tool accepts traces as generated. The cache simulation results can be plotted with various plotting tools (e.g. GnuPlot). Plotting the graphs is supplemented through scripts that parse cache simulation results.

4.2. Cache Behavior

To understand the usefulness of Gleipnir’s traces, we must touch on the subject of processor cache indexing. A processor’s cache is a small intermediate memory unit indexed using some of the address bits of the accessed object, the index bits normally rely on modulo arithmetic. In our previous work [18] we have shown that the cache accesses of most applications are non-uniform, causing some cache lines (or sets) to be underutilized while other cache lines are overutilized. The heavily accessed cache lines cause most of the cache conflicts and cache misses. Cache misses can be minimized if the cache accesses are distributed more uniformly. This can be achieved either
by changing how cache indexes work (requiring hardware modifications), or by changing the data layout of conflicting data structures. Using Gleipnir generated traces we can explore both research areas.

4.3. Cache simulation

Observing how an application uses cache memory requires cache simulators\(^3\). Trace-driven simulators take address traces as input, map each trace to a cache access, and identify if the access is a hit or a miss. The accuracy of the statistics regarding hits and misses and the level of detail about hits and misses provided depend on the capabilities of the cache simulators.

For our purpose, we modified a widely-used, albeit simple, cache simulator. We extended the simulator to accept Gleipnir traces and track cache hits and misses in greater detail. For example, we can track hits and misses per function, or each program variable including dynamically allocated structures. We report the cache statistics hierarchically: starting with summary results for the entire program at the highest level, to per program variable at the lowest level.

We illustrate our analysis using a MiBench benchmark, jpeg. We focus on cache accesses caused by the benchmark’s stack, heap, and global address

\(^3\)Other analysis techniques involve models, static analysis, and hardware performance counters.
--- Simulation begins.
2000000, 4000000, 6000000, 8000000
10000000, 12000000, 14000000
--- Simulation complete.

### Metrics

<table>
<thead>
<tr>
<th>Demand Fetches</th>
<th>Total</th>
<th>Data</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>14642192</td>
<td>14642192</td>
<td>11062461</td>
<td>3479731</td>
<td></td>
</tr>
<tr>
<td>Fraction of total</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.7607</td>
<td>0.2393</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Demand Misses</th>
<th>Total</th>
<th>Data</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>109144</td>
<td>109144</td>
<td>68824</td>
<td>40320</td>
<td></td>
</tr>
<tr>
<td>Demand miss rate</td>
<td>0.0075</td>
<td>0.0075</td>
<td>0.0062</td>
<td>0.0116</td>
</tr>
</tbody>
</table>

Table 3: Cache simulation overall results

spaces. When the simulation completes the first output shown is the overall cache statistic for every cache level as shown in Table 3. The user can then observe how well the cache is utilized. The user can then identify the functions of the application that are causing most cache accesses and misses. This data for jpeg is shown in Table 4; sorted by the number of accesses and misses. In addition to showing the function names, the data shows the number of accessed variables within these functions.

<table>
<thead>
<tr>
<th>Accesses</th>
<th>Hits</th>
<th>Misses</th>
<th>Miss %</th>
<th>Function / Segment</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>29993984</td>
<td>2909921</td>
<td>83943</td>
<td>2.80</td>
<td><em>HEAP</em></td>
<td>0</td>
</tr>
<tr>
<td>33584902</td>
<td>3315472</td>
<td>43668</td>
<td>1.32</td>
<td>encode_mcu_AC_refine</td>
<td>13</td>
</tr>
<tr>
<td>10878411</td>
<td>10859373</td>
<td>19038</td>
<td>0.61</td>
<td>encode_mcu_DC_refine</td>
<td>13</td>
</tr>
<tr>
<td>1148436</td>
<td>1129715</td>
<td>18721</td>
<td>1.29</td>
<td>forward_DCT</td>
<td>14</td>
</tr>
<tr>
<td>641088</td>
<td>633216</td>
<td>7872</td>
<td>1.23</td>
<td>encode_mcu_DC_first</td>
<td>10</td>
</tr>
<tr>
<td>116556</td>
<td>113546</td>
<td>3010</td>
<td>2.58</td>
<td><em>STACK</em></td>
<td>0</td>
</tr>
<tr>
<td>539206</td>
<td>536581</td>
<td>2625</td>
<td>0.49</td>
<td>emit_symbol</td>
<td>6</td>
</tr>
<tr>
<td>4480</td>
<td>2658</td>
<td>1822</td>
<td>40.67</td>
<td>_IO_file_xsgetn</td>
<td>2</td>
</tr>
<tr>
<td>1230919</td>
<td>1229140</td>
<td>1779</td>
<td>0.14</td>
<td>emit_bits</td>
<td>6</td>
</tr>
<tr>
<td>736464</td>
<td>734890</td>
<td>1574</td>
<td>0.21</td>
<td>compress_output</td>
<td>15</td>
</tr>
<tr>
<td>30444</td>
<td>28915</td>
<td>1549</td>
<td>5.08</td>
<td>encode_mcu_DC_refine</td>
<td>8</td>
</tr>
<tr>
<td>52256</td>
<td>50728</td>
<td>1528</td>
<td>2.92</td>
<td><em>GLOBAL</em></td>
<td>0</td>
</tr>
<tr>
<td>19560</td>
<td>18169</td>
<td>1391</td>
<td>7.11</td>
<td><em>GLOBAL</em></td>
<td>0</td>
</tr>
<tr>
<td>949296</td>
<td>948190</td>
<td>1106</td>
<td>0.12</td>
<td>jpeg_gen_optimal_table</td>
<td>14</td>
</tr>
<tr>
<td>1669916</td>
<td>1668950</td>
<td>966</td>
<td>0.14</td>
<td>jpeg_gen_optimal_table</td>
<td>14</td>
</tr>
<tr>
<td>66656</td>
<td>5767</td>
<td>869</td>
<td>13.36</td>
<td><em>GLOBAL</em></td>
<td>0</td>
</tr>
<tr>
<td>9728</td>
<td>8985</td>
<td>763</td>
<td>7.64</td>
<td>jpeg_write_scanlines</td>
<td>7</td>
</tr>
<tr>
<td>7937</td>
<td>7209</td>
<td>728</td>
<td>9.17</td>
<td>process_data_simple_sidebar</td>
<td>7</td>
</tr>
<tr>
<td>9984</td>
<td>9264</td>
<td>720</td>
<td>7.21</td>
<td>jpeg_write_scanlines</td>
<td>7</td>
</tr>
<tr>
<td>4401</td>
<td>3780</td>
<td>621</td>
<td>14.11</td>
<td>start_pass_phuff</td>
<td>9</td>
</tr>
<tr>
<td>8722</td>
<td>8196</td>
<td>526</td>
<td>6.03</td>
<td>rgb_ycc_start</td>
<td>6</td>
</tr>
<tr>
<td>11904</td>
<td>11390</td>
<td>514</td>
<td>4.32</td>
<td>rgb_ycc_start</td>
<td>6</td>
</tr>
<tr>
<td>8106</td>
<td>7672</td>
<td>434</td>
<td>5.36</td>
<td><em>GLOBAL</em></td>
<td>0</td>
</tr>
<tr>
<td>2886</td>
<td>2500</td>
<td>386</td>
<td>13.37</td>
<td>main</td>
<td>13</td>
</tr>
<tr>
<td>274176</td>
<td>273804</td>
<td>372</td>
<td>0.14</td>
<td>jpeg_make_c_derived_tbl</td>
<td>14</td>
</tr>
<tr>
<td>4096</td>
<td>3740</td>
<td>356</td>
<td>8.69</td>
<td>get_raw_row</td>
<td>5</td>
</tr>
<tr>
<td>8665</td>
<td>8420</td>
<td>245</td>
<td>2.83</td>
<td>jpeg_make_c_derived_tbl</td>
<td>14</td>
</tr>
</tbody>
</table>

Table 4: Cache simulation function output

From Table 4 we can see that three functions, `encode_mcu_AC_refine`, `encode_mcu_AC_first`, and `rgb_ycc_convert`, cause a large number of cache misses and they access 13 to 15 program variables. This information can be seen in lines 3, 5, and 6 of Table 4. We can also see that accesses to `heap` elements, line 2, are responsible for nearly 80,000 misses and 3 million accesses. Accesses to
stack elements, line 4, are considerably fewer with about 20,000 misses even though there are more than 10 million accesses to the stack area. Our cache simulator can be configured to provide statistics for selected execution intervals (for example for every n executed instructions or m cache misses). This type of information can reveal the dynamics of accesses to different address spaces and program variables during the program execution.

Figure 5 and Figure 6 show the number of accesses and misses during a program’s execution. The X axis is the time slice, i.e. the interval where data was accumulated (here for every 300,000 load, store or modify accesses). The Y axis is the percent of the 300,000 accesses to stack, heap of global address spaces. The figure allows us to observe how the accesses to these different address regions change over the life of a program execution. In this example it can also be observed that most of the accesses to stack are hits (very few misses), while most of the access to heap addresses are misses. One can then explore the causes of these accesses and misses.

![Graph showing data accesses percent ratios for every 300k load, store, or data modifies for Mibench jpeg benchmark.](image)

The simulator’s data is also capable of tracking the percent of total accesses and misses during the execution intervals to individual program variables (or set of variables). In this paper we did not include hit and miss data for individual variables.

For every function displayed in Table 4, the simulator keeps track of individual program elements as well as their accesses for each cache set. This is later displayed by detailed per cache set hits and misses table. Such information can be used to identify cache sets that are heavily accessed (and cache sets that are underutilized).

Assume we are interested in the `encode_msu_AC_refine` function. Table
Figure 6: Data miss percents for every 10k load, store, or data modify instructions for Mibench jpeg benchmark.

5 shows the number of hits and misses for each cache set that the function touched during its execution. It also identifies the number of accessed variables or structures during the function call. In the first column we can see the detailed output per cache set for the function. This is the total number of all accesses, hits, misses for all variables and structures in the function. In the middle and right column we can see some of the variables accessed in encode_mcu_Ac_refine. We present the variables and structures with the highest misses. The data for variables is by the number of misses; in our example, variable sizeofobject caused more than 31k misses. We have omitted some cache sets for illustration purposes. The 3rd column shows data for some other variables, absvalues, jpeg_natural_order, block, Al, cinfo, etc.

Notice that every variable shown in our tables is annotated by its scope. For example sizeofobject is identified by an H indicating that this variable is allocated on heap, absvalues by LS indicating that the variable is a local structure, or jpeg_natural_order with GS indicating that the variable is a global structure. This helps locate the variable in the source code and reason about its scope.

4.4. Visualizing data layout

The data provided by the simulator is formatted so the data can be easily plotted. We relied on a set of scripting modules to produce gnuplot friendly output files. This allows us to visualize the data layout. Figure 7 plots the function encode_mcu_Ac_refine.

The X axis is the cache set number, and the Y axis is the number of hits, and misses to each cache set. The top graph shows the number of hits and the bottom graph shows the number of misses. We can reason that cache
areas with most hits are likely to exhibit most misses, as can be seen from the graphs. In this example the simulation used a 32 kilobyte, direct mapped L-1 data cache with 32 bytes per block and 1024 sets. The function shows most misses for cache sets around 175, 450, 725, 975-1023. From the graph we can see that the dynamically allocated variable `sizeofobject` touches every set in the cache. The spikes come as a result of contention between this variable and other variables such as `absvalues`, `jpeg_natural_order`, `block`, `Al`, or `cinfo`.

There can be many reasons for cache misses and Gleipnir along with our cache simulator allows us to observer memory accesses and cache hits and misses in great detail. The per set statistics (shown in Figure 7) can be used to develop a cache cost model which can be used by compilers. For example we have used cost based on evictions of variables and identify the variable that is causing most evictions of (or conflicts with) other variables.

Table 6 shows the functions and variables that have caused the most evictions in function `encode_mcu_AC_refine`. For example consider the dynamic structure `sizeofobject`; most evictions (24k out of a total of 31k misses) of this variable are caused by other variables or structures of the same function (Table 6). Also notice that of the total 24k evictions almost 16k are from accesses to the same structure, line 2. About 6k misses are from accesses to
absvalues variable, and the rest by other data. Calls to other functions, e.g. encode_mcu_DC_refine, is the reason for nearly 2.5k misses combined. This information has two implications. Firstly, the cache size is simply too small to hold the entire dynamic structure allocated from sizeofobject resulting in sizeofobject evicting itself and secondly, perhaps a different structure layout may remedy the number of misses. For example frequently used structure objects need to be pinned to specific cache sets through custom memory allocation techniques.

5. Future Work

Our goal in developing Gleipnir is to make program analyses and profiling techniques easy to use by average programmer, and by combining Gleipnir traces and cache simulation we aim to expose the underlying data conflicts that may occur between various program data-structures. The ability to account for common cache performance bottlenecks is not an easy problem,
and even more difficult is to provide an efficient method for data placement analyses and optimizations. Our future work aims to bridge some current shortcomings with our cache simulator.

5.1. Multi-core and multi-process analysis

Multi-core and multi-process cache analyses are limited by our current simulator. Private level caches are virtually indexed. Meaning that a data’s virtual address is sufficient to place an element into the cache; however, shared caches and last level caches (LLC) are physically indexed; this requires us to generate physical addresses, unlike the examples shown thus far which use virtual addresses.

The physical address is assigned by the operating system when a virtual page is mapped to a physical page and thus unavailable to most instrumentation frameworks and tools. However, Gleipnir includes a mechanism that takes advantage of operating system kernel maps and pagemap files to extract the physical addresses.

Table 6: Cache simulation’s cost-matrix for function encode_mcu_AC_refine.

---

4We are not aware of any other instrumentation frameworks or plug-in tracing tools that are capable of mapping physical and virtual addresses.

---
When physical address tracking is enabled Gleipnir will track allocated virtual and physical pages. The trace in Listing 9 shows an example trace of the code in Listing 1 with physical address tracking enabled. The trace looks similar to the basic Gleipnir trace line except that the physical address is added to the trace line.

Furthermore, to achieve a more accurate shared cache simulation we have to account for cycle information, so that accesses from multiple processes or threads can be interleaved. The current cache simulator is trace-driven cycle-unaware but a future cache simulation model can rely on instruction cycle information.

Fetching instruction types is trivial using the Valgrind framework. By using our tracing tool in combination with instruction cycle models (providing number of cycles needed for the instructions) and physical addresses we can achieve the desired simulation of accesses to shared data structures and the accesses to shared caches accurately.

5.2. Identifying logical structures

Currently Gleipnir will track single structure allocations and group them by its name. For example multiple allocations to structure xyz are enumerated as xyz1,2,3...etc. We use the term logical structures to identify dynamic abstract datatypes such as linked lists or trees. The goal is to identify logical structures (to emulate the linked list created during a program executing) and using cache simulators to identify conflicts among different elements of a
linked list or between different linked lists. Or we can identify which fields of a structure are accessed frequently when traveling a linked list. The ability to identify logical structures may help a user to identify the potential cache benefits of utilizing various structures (e.g. linked lists over trees).

Listing 10: Example source code

```c
int my_gl, my_glArray[10];
struct type {
  int A;
  int B[10];
};
struct type GStruct[2];

void functionA(int param1) {
  int fnc_local;
  param1 = 12345;
  fnc_local = 1234;
  my_gl = 543;
  my_glArray[2] = 123;
}

int main(void) {
  GLEIPNIR_START_INSTRUMENTATION;
  functionA(main_local);
  my_gl = 234;
  my_glArray[2] = 123;
  GStruct[1].A = 5;
  GStruct[0].B[3] = 123;
  GLEIPNIR_STOP_INSTRUMENTATION;
  return 0;
}
```

Listing 11: Trace with values enabled

```
START PID 30773
S 7 ff000160 8 1 S main
S 7 ff00016c 4 1 S main LV main_local [0x64]
L 7 ff00016c 4 1 S main LV main_local [0x64]
S 7 ff000120 8 1 S main
S 7 ff000118 8 1 S functionA
S 7 ff000104 4 1 S functionA LV param1 [0x64]
S 7 ff000110 4 1 S functionA
S 7 ff000114 4 1 S functionA LV fnc_local [0x4d2]
S 000601098 4 1 G functionA GV my_gl [0x21f]
S 0006010a8 4 1 G functionA GS my_glArray [2] [0x7b]
L 7 ff000118 8 1 S functionA
L 7 ff000120 8 1 S functionA
S 000601098 4 1 G main GV my_gl [0x6a]
S 0006010a8 4 1 G main GS my_glArray [2] [0x7b]
S 00060106c 4 1 G main GS GStruct [1].A [0x65]
S 000601050 4 1 G main GS GStruct [0].B [3] [0x7b]
S 7 ff000130 8 1 S main LS _zzq_args [0]
S 7 ff000140 8 1 S main LS _zzq_args [2]
S 7 ff000148 8 1 S main LS _zzq_args [3]
S 7 ff000150 8 1 S main LS _zzq_args [4]
S 7 ff000158 8 1 S main LS _zzq_args [5]
S 7 ff000112c 4 1 S main
L 7 ff000112c 4 1 S main
```

We can achieve this by tracking pointer values used when an allocation occurs and keeping a list of active pointers. By utilizing user input parameters to identify pointer types within defined structures we can track and combine multiple structure allocations into unified logical structures. For example several allocations to structure `xyz` can be linked into a list `xyz`.

Our current implementation can identify values at address locations by enabling the `-track-values=yes` flag for Gleipnir. The trace snippet in Listing 11 shows an example of traced values, in hex format, when Listing 10 is executed.

5.3. Trace driven data-structure transformations

Another potential use for Gleipnir as demonstrated in [19] is to utilize the traces for semi-automatic trace-driven data-structure transformations. Observing various structure transformations is important to reason about potential cache effects different structure layouts may have. By utilizing Gleipnir’s
traces we implemented a trace-driven module that semi-automatically transforms data-structures and analyzes the cache effects. It is semi-automatic because it relies on the user to supply transformations according to rules and automatic because the module can identify the transformation rules and automatically apply them on the trace.

6. Conclusions

In this paper we described a tracing and profiling tool called Gleipnir. Gleipnir is built using a well-known dynamic binary instrumentation tool, Valgrind. We elaborated on how Gleipnir works in combination with our cache simulator. We have illustrated how Gleipnir traces programs and how to interpret the output generated by Gleipnir. Gleipnir provides very detailed information on every memory access including the name of the variable accessed, if the variable is a static, global or dynamically allocated element, the function that contained the access and the thread that executed the access. For dynamically objects (or heap objects), Gleipnir tracks multiple allocations with the same structure name, and the element within the structure that was accessed.

We demonstrated the ability to produce detailed application cache analysis as well as a cost-matrix (Table 3, 4, 5, and 6) identifying the source of cache misses. We have shown how to track which variables are evicting a particular variable. Such information is valuable in exploring different data layouts, code and data refactoring techniques (such as tiling, fusing etc.).

We are currently extending Gleipnir to produce both virtual and physical address with memory accesses so that shared objects in multithreaded applications can be tracked and the performance of shared caches and last level caches can be studied.

Acknowledgements

This work is made possible in part by support from the NSF Net-Centric Industry/University Cooperative Research Center, a grant from Advanced Micro Devices, and ORNL summer internship support. We would like to thank our former and current undergraduate students Brandon Potter and Eric Schluter. The name Gleipnir is taken from Norse mythology and refers to a deceptively strong ribbon that was used to restrain a vicious wolf called Fenrir. Gleipnir may help programmers to restrain the memory hogs in applications.
References


