# Scalability Analysis of Gleipnir: A Memory Tracing and Profiling Tool, on Titan

Tomislav Janjusic*, Christos Kartsaklis*

* Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831-6173
Email: {janjusict,kartsaklisc}@ornl.gov

Wang Dali†

† Climate Change Science Institute
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831-6173
Email: wangd@ornl.gov

***Abstract*—Understanding application performance properties is facilitated with various performance profiling tools. The scope of profiling tools varies in complexity, ease of deployment, profiling performance, and the detail of profiled information. Specifically, using profiling tools for performance analysis is a common task when optimizing and understanding scientific applications on complex and large scale systems such as Cray's XK7. Gleipnir is a memory tracing tool built as a plug-in tool for the Valgrind instrumentation framework. The goal of Gleipnir is to provide fine-grained trace information. The generated traces are a stream of executed memory transactions mapped to internal structures per process, thread, function, and finally the data structure or variable. This paper describes the performance characteristics of Gleipnir, a memory tracing tool, on the Titan Cray XK7 system when instrumenting large applications such as the Community Earth System Model.**

## I. Introduction

For many applications processor and memory speed is still a major performance bottleneck. In order to reduce the speed-gap application developers must carefully consider program data-structure layout, data placement, and application data-flow. Therefore, in order to make intelligent design and development choices the programmers use performance profiling tools and analysis software to gain insights into their application's behavior. The realm of application profilers is vast and the tools vary with respect to their complexity, easy of deployment, profiling performance, and the detail of profiled information. Moreover, tools focus on specific performance metrics, as a consequence developers can combine several tools to gain a complete and accurate application's behavior representation.

Broadly speaking, application profiling tools can be categorized based on their data gathering methodologies into event-driven, statistical, and instrumenting. Instrumenting tools can be further sub-categorized into compiler assisted, binary translation, binary instrumentation, and hybrids or runtime code manipulation tools. From a users' perspective, instrumentation tools are software that manipulate an application's code by injecting foreign code at interesting locations of an application's source code or executable. When the injected code executes it records or otherwise collects application's data limited by the frameworks' scope. We must differentiate the instrumentation tool and the instrumentation framework. In this article we will refer to the framework as the underlying mechanism which enables the development of plug-in tools to instrument an application. For example Gleipnir[1] is a plug-in tool for the Valgrind[2] instrumentation framework. Similarly other instrumentation frameworks provide a set of plug-in tools to profile application's performance metrics.

Various frameworks enable the development of fine-grained instruction and memory tracing. The general rule is that exposing a greater detail implies a significant performance overhead, and more importantly recording this detail incurs additional space and time overhead. When fully enabled Gleipnir provides instruction level load, store, and modify data traces with debug information to function call, data-type, data-scope, and thread information. In addition, dynamic memory allocations can be intercepted and their accesses tracked back to the source code file and line-number, or a manually instrumented name.[1] 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, and the analysis of the traces is external to the tool. While the plug-in tool can provide some basic information about the application's runtime behavior the user must further analyze Gleipnir's traces to get a more meaningful picture. For example during instrumentation the tool can gather various instruction and process related information such as: the total number of instructions executed, the total number of data reads, writes, or modifies, the total number of function entries, average stack size, total number of dynamically allocated blocks,average memory usage, etc. To analyze the cache behavior we must rely on CPU cache simulators. Gleipnir provides a modified, albeit simplistic, cache simulator DineroIV[3] for this purpose but the user is free to use other simulators provided that the trace is compatible. The traces should provide the necessary



[1]A user may want to instrument their application manually which can be a more descriptive name for dynamically allocated blocks.

trace detail on every data memory access and map the access to source code variable or manually instrumented identifiers. This information can be used to deduce a variety of conclusions about an application's data-structure behavior, such as the structure's avarage life-time, average number of accesses, average offset, or data-structure hot spots. Will we discuss the internal mechanism in subsequent sections.

The rest of the article is organized as follows: in Section II we will discuss related work and touch on a few similar frameworks which may be used to develop similar plug-in tools. In Section III we will explain how Gleipnir works and provide a trace example. In Section IV we compare Gleipnir's performance with various tracing options and compare the performance characteristics. In Section V we will summarize our conclusions and talk about the current tool status.

## II. Related Work

We explained in Section I that application tuning and optimization is supplemented through application profiling tools. Although application profiling is mostly tool driven it is not uncommon for programmers to manually profile an application by inserting code fragments to provide additional information about the program's state. The simplest form, and sometimes quite sufficient, is inserting *printf()* statements.

Valgrind's framework comes with a set of widely used tools. The tool that Valgrind is most known for is Memcheck, a memory leak dectector, but it also provides other tools geared towards profiling such as Cachegrind[4], Callgrind[5], and Massif. Cachegrind is a cache-simulation tool that provides information about the application's cache behavior. Cachegrind can utilize Valgrind's debug parser and relate collected cache statistics with source-code line numbers. Callgrind, a call-graph profiler, is based on Cachegrind and provides a profiled call-graph annotated with cache performance information. The tool is supplemented with an advanced graphical user interface, KCachegrind, to help visualize Callgrind's information. Massif is a heap profiler tool that can analyze an application's heap usage which helps the user to analyze the application's memory regions and determine overall heap utilization.

A similar and performance efficient tool is Pin[6], a dynamic binary instrumentation framework which closely follows the model of the popular ATOM tool[7]. Pin allows the plug-in tools to inject code dynamically at runtime. The key difference between Pin and Valgrind is that Pin does not simulate the application's instructions, but using a different method controls and instruments the application code. Pin comes with various tracing and plug-in tools such as, pinatrace, Maid, and other tools which provide basic information on data blocks, instruction, and memory traces.

Within the same category we find DynInst[8] designed for code patching and performance measurement. Similar to Pin and Valgrind , Dyninst can apply instrumentation at runtime and insert instrumented code at arbitrary points. Moreover, the ability to insert arbitrary analyses routines makes Dyninst a good framework for large scale scientific program where the granularity of collected data needs to be adjusted. Several other performance tools were built either utilizing the Dyninst framework, or built around on the same model.

DynamoRIO [9] stands for dynamic introspection, instrumentation, and optimization. It is an instrumentation framework which allows building of dynamic profiling tools and allows the user to dynamically modify existing binaries to improve application's performance. Similar tools built using the framework are TaintTrace[10], Adept[11], Dr. Memory, etc.

Virtually all modern CPUs come with a number of hardware performance counters. Performance counters are hardware units which populate registers with performance data during various hardware events. For example when a cache miss occurs an L1-cache hardware counter will increment, or when a memory-bus transaction happens a different counter will increment. An application can interupt the system and flush a number of desired hardware performance counters. Accessing hardware performance counters too often will preturb the application and the result may be skewed. Therefore tools or users must take care to control the granularity of accessing performance counters. For most performance profiling application accessing performance counters is facilitated using libraries such as PAPI[12]. Performance profiling through hardware counters is arguably the most common way of collecting trace information and other statistics. Hardware counters are particularly a very practical way of application profiling because of the low performance overhead. Some tools which use the PAPI library are TAU[13], HPCToolkit[14], OpenSpeedShop[15], etc.

The various performance profiling tools, particularly instrumentation tools incur a heavy overhead; however, the overhead is offset by the level of detail that instrumentation based profilers and tracing tools can collect. It was reported in [6] and [2] that the average slowdown for a basic block count application built in Valgrind is $8\times$, about $2.5\times$ for Pin, and $5.1\times$ for DynamoRIO.

## III. Gleipnir Overview

### A. Valgrind's IR

As mentioned in the introduction Gleipnir's underlying framework is Valgrind. Valgrind consists of a core-tool and plug-in tools. The core-tool operates on sections of code blocks known as *SuperBlocks (SB)*. An SB is a single-entry multiple-exits block, composed of multiple basic-blocks (single-entry single-exit) consisting of roughly 50 instructions. Note that Valgrind's plug-in tools are statically compiled with the core-tool. The Valgrind core-tool and plug-in tool interaction is shown in Figure 1. An SB is transformed into an Intermediate Representation (IR) which is passed to the plug-in tool. During the transformation a single instruction is disassembled into multiple intermediate instructions. The IR is instrumented by the plug-in tool and passed back to the core-tool for re-synthesis. The IR is recompiled into machine-code and executed on a simulated CPU. This is an important distinction of Valgrind and other tools. The native application's code never touches the host architecture. The interaction of the

core-tool and plug-in tool is mostly abstracted through a rich set of API calls through which the plug-in tool can instrument and make core-tool requests.

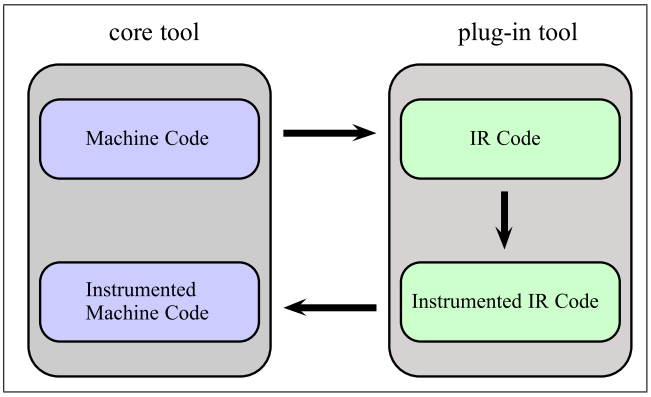


Fig. 1: SuperBlock flow chart

### B. Tracing Instructions

Gleipnir's main functionality operates on instruction events provided by Valgrind. From Gleipnir's perspective an event is either an instruction read (Ir), data read (Dr), data write (Dw), or data modify (Dm).[2] During program instrumentation Gleipnir will parse the incoming SB and insert *dirty helper calls* at every instruction iteration. The helper function can record any number of events, but most are related to instruction's address look-up, debug information annotation, and basic instruction counting. A subset of other helper calls will call other aspects of Gleipnir's functionality. These include techniques to determine changes in application's stack, change in application's thread id, function entries, various events related to changes in program instrumentation granularity, etc. The basic helper calls are *trace_instruction, trace_data_read, trace_data_write, and trace_data_modify*. This means that at each instruction the helper call will incur a function call, and depending on the tracing granularity proceed to parse various debug symbols and recorded data blocks using the instruction address. We must differentiate between two similar but different mechanisms of handling instruction to source code variable mapping.

*1) Local data:* If an application's binary image retained the debug information Gleipnir can use Valgrind's debug information to parse the internal debug table. Enabling this feature implies that we must also enable Valgrind's option *read-var-info* which will track stack changes and load the necessary debug symbols. Note that enabling this option under Valgrind will incur a heavy overhead and render the tool impractical for large running applications.

*2) Dynamic and Global data:* Gleipnir can wrap or replace *malloc()* calls. Wrapping implies that a malloc call is intercepted, executed, and recorded while replacing implies that the malloc call is intercepted and redirected. Valgrind's basic mechanism involves malloc replacement to an internal Valgrind malloc implementation. Thus from Gleipnir's perspective this provides a mechanisms to intercept and record every dynamic allocation. The goal is to intercept and tag dynamic memory with a descriptive id, and subsequently annotate any instruction access which references these blocks. Note that dynamically allocated blocks are not associated with any debug symbols, after all they are chunks of randomly allocated memory. Therefore, in order to trace instruction references to dynamic blocks Gleipnir provides two mechanisms to record and annotate these accesses.

The first is to parse the application's stack, and find the instruction pointer which executed the malloc routine. The tool can use the instruction's address and find the related debug information. This usually implies that we can annotate the allocated block with the originating file name and line number. For simple malloc routines, which have only a single callee this will suffice. However, we often find that malloc routines can be buried deep in the call stack and thus it becomes impossible to determine the originating function. This may result in incorrect debug annotation and may distract from correctly determining the data-structures that are referenced or otherwise accessed in the instruction trace.

To improve the instruction to dynamic memory mapping, Gleipnir provides a set of client interface calls that allow the user to annotate blocks manually. In this terminology the client is the application being instrumented, thus client interface calls are macro routine which requires the user to provide a descriptive string. The string is buffered and used as a tag for the upcoming malloc call. Similarly, if a user wants to modify the name of an already allocated block he can invoke similar client interface calls to update or otherwise modify a blocks name. The difference is that these calls require that the user passes a descriptive string as well as the address referencing the allocated block. Note that the reference address must not necessarily be the base address because internally Gleipnir will search recorded block address ranges.

Users can also use client interface calls to annotate global data-structures. Similar to dynamically allocated blocks, annotating global data-structures requires that the user passes a string name to tag the block as well as the base address and the size of the structure. For most practical purposes annotating dynamic and global data will suffice to capture and obtain an accurate picture of an application's overall data access patterns. In the following subsections we will elaborate on the tool's various options and available user client interface calls.

### C. Trace examples

In Figure 2 and Figure 3 and 4 we show an example source code and resulting trace examples, respectively. These figures are for illustration purposes to demonstrate what the end-user may expect to observe when tracing larger applications. Note that even small applications may generate data files several gigabytes large. Therefore, a user must be careful when enabling and disabling instrumentation points.

[2]The events are architecture host/target specific thus different architectures may involve different events.

Figure 2 shows a toy program already instrumented. Every application must include the *gleipnir.h* header file, which contains all client interface macros. In this example we will request that the malloc call be annotated with the *mystruct* keyword, record the global data-structure with the *myGlobal* tag, and we start and stop the instrumentation after a few assignment statements and a function call. When we execute the application with Gleipnir we will see the resulting trace shown in Figure 3 or Figure 4 depending on the instrumented level of detail.[3] This example traces shows the typically encountered trace information. In Figure 3 we can observe the trace only with user annotations, and in Figure 4 we can see the full trace with debug symbols and user annotations.

```
#include <stdlib.h>
#include "../valgrind-trunk/gleipnir/gleipnir.h"

typedef struct _type{int Ab; int Ba;} mytype;
mytype myG;

int foo(int f_loc)
{
 f_loc+=myG.Ab;
 return f_loc;
}

int main(void)
{
  int A = 0;
  int Arr[10];

  GL_RECORD_GLOBAL("myGlobal", &myG,
                   sizeof(mytype));

  GL_RECORD_MSTRUCT("mystruct");
  int* ptr = malloc(sizeof(int) * 50);

  GL_START_INSTR;
    A = 123;
    Arr[5] = A;
    myG.Ab = 7; myG.Ba = 2;
    *(ptr+25) = foo(Arr[5]);
    ptr[5] = 10;
  GL_STOP_INSTR;
  return 0;
}
```

Fig. 2: An example source file.

There are two types of trace-lines: regular and keywords. The first symbol identifies if the trace-line is a keyword or a regular access. A regular access type can be a Load, Store, Modify, or Instruction (L,S,M, or I). A keyword trace-line is denoted as an X. Keyword instructions are a special type of Gleipnir inserted trace-lines that describe various events that happen during tracing. For example *X START 0:15401 at 0* indicates that this is the first trace file whose parent PID is 0 and its PID is 15401 starting at instruction 0, or *X 1 MALLOC 005188030 200 mystruct 0* indicates that the program with thread id 1 allocated a 200 byte memory block at address 0x005188030 named *mystruct* and this is the first instance of that structure. Users can also insert custom keywords. The main goal of keywords is to have a descriptive tag which may aid simulation or analysis tools.

In regular trace-lines the second field is the data's *virtual address* followed by its *access size*. The fourth field is the data's originating *thread id* followed by the originating *segment* which can be stack, global, or heap (s,G, or H). All instructions adhere to these five basic fields; however, if the access references a dynamic, static, or otherwise mapped memory region, then the trace-line will contain additional debug information. The additional debug information is: the data's originating *function*, its *access scope*, and finally the *variable or structure name* including its *access offset*. As an example in Figure 3 we have 3 accesses to a global structure (GS), *myGlobal*, at offset 0 and offset 4, and 2 accesses to a heap block zero (H-0), *mystruct*, at offset 100 and offset 20.[4]

```
X START 0:15401 at 0
X THREAD_CREATE 0:1
X 1 MALLOC 005188030 200 mystruct 0
S ffeffd0c8 8 1 S main
S ffeffd11c 4 1 S main
L ffeffd11c 4 1 S main
S ffeffd0f4 4 1 S main
S 000601030 4 1 G main GS myGlobal.0
S 000601034 4 1 G main GS myGlobal.4
L ffeffd110 8 1 S main
L ffeffd0f4 4 1 S main
S ffeffcff8 8 1 S main
S ffeffcff0 8 1 S foo
S ffeffcfec 4 1 S foo
L 000601030 4 1 G foo GS myGlobal.0
M ffeffcfec 4 1 S foo
L ffeffcfec 4 1 S foo
L ffeffcff0 8 1 S foo
L ffeffcff8 8 1 S foo
S 005188094 4 1 H main H-0 mystruct.100
L ffeffd110 8 1 S main
S 005188044 4 1 H main H-0 mystruct.20
S ffeffd090 8 1 S main
S ffeffd098 8 1 S main
S ffeffd0a0 8 1 S main
S ffeffd0a8 8 1 S main
S ffeffd0b0 8 1 S main
S ffeffd0b8 8 1 S main
X INST 1136
X END 15401 at 1136
```

Fig. 3: An example trace file without debug info.

Because Gleipnir relies on Valgrind's internal debug parser to parse the debug information for local as well as global data, any application that needs to be profiled for static data must be compiled with the compiler's *-g* flag, and executed with enabling Valgrind's *read-var-info* flag as well as enabling Gleipnir's *read-debug* flag. Enabling the read debug flag allows to capture a very descriptive set of traces at the cost of much higher performance overhead. Figure 4 is an example trace when the application was executed with the debug flags

[3]Usually the trace-file is very large, even for simple applications due to library initialization, thus for illustration purposes we have suppressed much of the trace information.

[4]Note that Heap block 0 simply means that this is the first instance of this particular structure. For example multiple allocations from the same structure will be tagged with the same name-tag but incremented instance counter.

on. We can observe more detailed debug information for the executed application. Instructions, which referenced local variables as well as global data have a more descriptive scope and variable information. For example using the debug symbol we can track variable and structure information up to nested individual elements, indicated with the trace-line, *S 000601034 4 1 G main GS myG.Ba*, shows a store instruction to the global segment from function *main* to the global structure *myG* and element *Ba*.

```
X START 0:15548 at 0
X THREAD_CREATE 0:1
X 1 MALLOC 005188030 200 mystruct 0
S ffeffd0c8 8 1 S main
S ffeffd11c 4 1 S main LV A
L ffeffd11c 4 1 S main LV A
S ffeffd0f4 4 1 S main LS Arr[5]
S 000601030 4 1 G main GS myG.Ab
S 000601034 4 1 G main GS myG.Ba
L ffeffd110 8 1 S main LV ptr
L ffeffd0f4 4 1 S main LS Arr[5]
S ffeffcff8 8 1 S main
S ffeffcff0 8 1 S foo
S ffeffcfec 4 1 S foo LV f_loc
L 000601030 4 1 G foo GS myG.Ab
M ffeffcfec 4 1 S foo LV f_loc
L ffeffcfec 4 1 S foo LV f_loc
L ffeffcff0 8 1 S foo
L ffeffcff8 8 1 S foo
S 005188094 4 1 H main H-0 mystruct.100
L ffeffd110 8 1 S main LV ptr
S 005188044 4 1 H main H-0 mystruct.20
S ffeffd090 8 1 S main LS _zzq_args[0]
S ffeffd098 8 1 S main LS _zzq_args[1]
S ffeffd0a0 8 1 S main LS _zzq_args[2]
S ffeffd0a8 8 1 S main LS _zzq_args[3]
S ffeffd0b0 8 1 S main LS _zzq_args[4]
S ffeffd0b8 8 1 S main LS _zzq_args[5]
X INST 1136
X END 15548 at 1136
```

Fig. 4: An example trace file with debug info.

### D. Gleipnir's flags and options

Users can use several options to control instrumentation detail and instrumentation speed and use the mentioned client interface macros to control instrumentation at run-time. The list of currently supported options is shown in Figure 5. The *fast-forward* option disables any instrumentation by passing every *SB* back to the core-tool. Users can enable full program instrumentation without any manual instrumentation using the *trace-state-on* flag; however, because of the timing overhead this option may be benefit only smaller test programs. Flags such as, *enable-parsing, prog-lang, trace-malloc-calls* control Gleipnir's automation with respect to malloc calls. It can parse the malloc call and automatically assign a unique structure name, we can also print every malloc routine or omit them because in large running application's these can happen very often. Several flags act like hints to control performance overhead: for example *multi-process, multi-threaded or is-mpi* will control the level of added instrumentation to check for application flow changes. Gleipnir is also able to track datum's physical address with the *map-phys, and track-pages* flags; however, this option is dependent on operating system capabilities because the necessary O.S. kernel modules must be present.

```
--fast-forward-on=no|yes
--trace-state-on=no|yes
--read-debug=no|yes
--enable-parsing=no|yes
--prog-lang='C'|'F'
--multi-process=no|yes
--multi-threaded=no|yes
--map-phys=no|yes
--track-pages=no|yes
--trace-instructions=no|yes
--trace-malloc-calls=no|yes
--trace-values=no|yes
--flush-at=(int)
--out-file=<filename>
--is-mpi=no|yes
```

Fig. 5: Gleipnir options.

Gleipnir's client calls allows the user to manually instrument the application. A user can turn on or off various instrumentation detail, or insert keywords into the trace. Other macros will allow the user to manipulate the recorded blocks or arbitrarily mark regions of memory and intercept any access to that address range.

- *GL_FAST_FORWARD_ON*
- *GL_FAST_FORWARD_OFF*
- *GL_GLOBAL_START_INSTRUMENTATION*
- *GL_GLOBAL_STOP_INSTRUMENTATION*
- *GL_START_INSTRUMENTATION*
- *GL_STOP_INSTRUMENTATION*
- *GL_MARK*
- *GL_MARK_STR*
- *GL_UPDATE_MSTRUCT*
- *GL_RECORD_MSTRUCT*
- *GL_UNRECORD_MSTRUCT*
- *GL_RECORD_GLOBAL*
- *GL_UMSG_STR*
- *GL_RENAME_TRACE*

Because every client call is part of the instrumented application, its instructions will end up in the final trace, thus the necessary global start and stop calls are implemented to allow users to pass multiple client interface calls without perturbing the trace. Sometimes it is important to understand various code sections, for that reason users can pass *GL_MARK, GL_MARK_STR* client calls that will insert miscellaneous keyword instructions into the trace for easier trace analysis. These keywords are later interpreted by the simulator for various internal tracking purposes. When debug information is not available or when users are only interested in dynamic and global structures: *GL_RECORD_MSTRUCT, GL_UNRECORD_MSTRUCT, GL_RECORD_GLOBAL* client calls serve to manually record memory blocks. An example of the manual instrumentation is shown in our example trace where the malloc call is recorded and traced as the *mystruct*

object which allows the tool to trace an access to the object's memory and annotate the instruction with the necessary debug information.

## IV. Scalability Analysis

### A. Experimental setup

We conducted our scalability and performance analysis using ORNL's Cray XK-7 Titan system. Titan consists of 18,688 compute nodes, each compute node is a 16-core AMD Opteron processor (Interlagos) with 32GB of physical memory. The operating system is *Cray Linux Environment* and the compute nodes use the *Compute Node Linux* micro-kernel. We chose the LAMMPS[16], Large-scale Atomic/Molecular Massively Parallel Simulator, application as our benchmark because of its ease of deployment and scalability. LAMMPS is a classical molecular dynamics code that models an ensemble of particles. The programming language of LAMMPS is C++. Our experiments were conducted across several nodes using a combination of Gleipnir options. Note that the tool was never tested on larger applications using hundreds of processing elements, thus the purpose of this work is to test the tracing tool's ability to handle real-world scientific applications.

### B. Timing comparison

The basic timing runs were conducted using three different setups. We compared the native run against the Valgrind's Nullgrind tool, and Gleipnir in *fast-forward* mode. The *fast-forward* mode is similar to Valgrind's basic tool, Nullgrind. Nullgrind is a tool used to test Valgrind's VEX library and core-tool functionality. This means that no instrumentation takes place and the SuperBlock (SB) is passed to the tool and immediately returned to core-tool for execution. During this process no instrumentation occurs and Gleipnir's traces contain only recorded events, such as number of executed instructions, events related to tracking memory regions, or user annotated events.

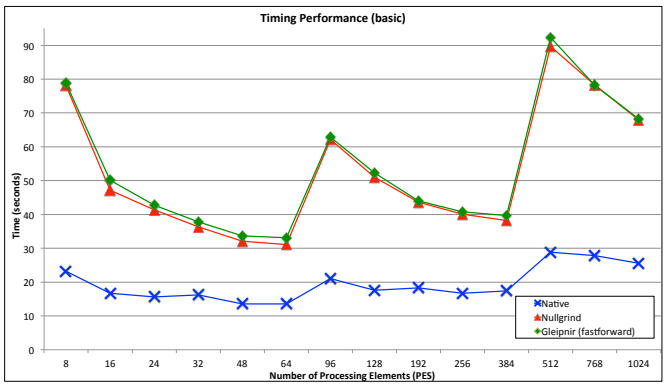


Fig. 6: Execution time comparison

Figure 6 shows the timing results when Gleipnir runs without any instrumentation enabled. The X-axis shows the number of processing elements and the Y-axis shows the overall execution time in seconds. For scalability we used varying problem sizes and up to 1k processing elements. We can observe that as we increase the number of cores the application's execution time improves, thus we doubled the problem size at 96 and quadrupled 512 processing elements. It was reported in [2] that Valgrind's overhead is about $\times 4$ which is more conservative than what we observed. Our simulation runs show that Valgrind's Nullgrind timing overhead is about $\times 3$ on average, and that Gleipnir's *fast-forward* mode is slightly slower. This implies that instrumented application's that use the *fast-forward* option to speed-up instrumentation in omitted code sections will run at most $\times 3$ slower than the native run.

### C. Memory usage

Valgrind's memory overhead is approximately one extra bit per byte. Various plug-in tools will add additional overhead depending on their functionality. Gleipnir's primary memory overhead is dictated by the application's memory allocation pattern. For example, for the same amount of memory a coarse allocation pattern will allocate fewer blocks which means that the tool's internal tracking mechanism will have fewer memory regions to track. On the other hand, a fine-grained memory allocation pattern will have more allocations thereby increasing the number of memory regions to track. The general rule is that for every allocation Gleipnir will allocate an additional 168 bytes. The data structure contains the chunks base address, size, name, and the number of accesses. The internal tracking is implemented similarly across several Valgrind tools. The malloc replacement function is triggered during an allocation and the control is redirected to the tool. Usually the tool will record the allocation request, request a malloc call on behalf of the application, and return the base address. Note that the malloc tracking mechanisms is the most significant memory overhead component. To compare the memory overhead we measured the virtual and resident memory usage with increased number of processing elements.

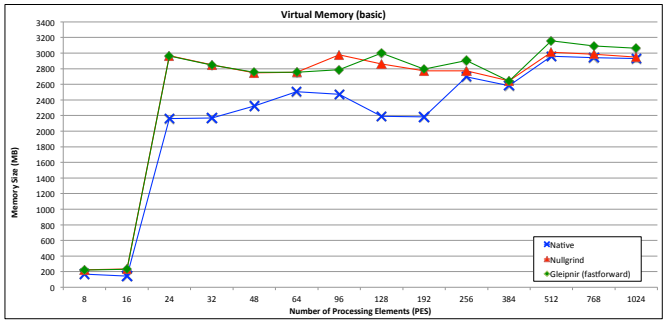


Fig. 7: Virtual memory usage comparison.

Figure 7 shows the average allocated virtual memory as the number of processing elements increases. The X-axis is the number of processing elements and the Y-axis shows the number of allocated memory in megabytes. It is somewhat surprising that: 1) on a single node (up to 16 PEs) the allocated virtual memory overhead is relatively low, and 2) the virtual memory increases dramatically when using more than 16 PEs, that is to say more than 2 nodes. Overall the Nullgrind

and Gleipnir will use about 20−30% more virtual memory. Gleipnir will incur a slightly larger overhead than Nullgrind, this is in part due to tracking events regardless of the tool's tracing mode.

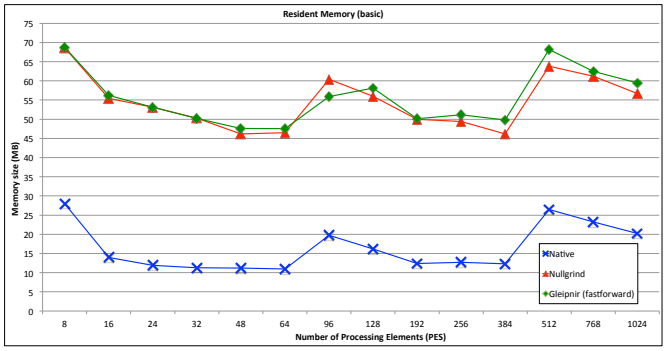


Fig. 8: Resident memory usage comparison.

Though the virtual memory behavior is somewhat puzzling, the resident memory as reported by our memory-daemon is on par with what we expect. The resident memory overhead is approximately $2-4\times$ of the native run. Notice that the resident memory diminishes as the number of processing elements increases. This is expected because we are not growing the problem size linearly, rather, as with the timing runs we increased the problem size at 96 and 512 PEs.

### *D. Effects of tracing and I/O overhead*

Enabling tracing reduces performance orders of magnitude, thus one must be very careful when choosing instrumentation points. Traditionally the tool's output was an ASCII file. This format was sufficient for smaller applications and for observing smaller code regions. Moreover, it did not require $3^{rd}$ party tools to view the trace. However, large application runs can easily output several gigabyte trace data. Therefore, to improve performance and save disk space we implemented a binary output mode.

```
typedef struct _binout_t{
  Addr    addr;
  UInt    instance;
  UInt    offset;
  UShort  func_id;
  UShort  var_id;
  UChar   atype;
  UChar   size;
  UChar   thread_id;
  UChar   segment;
} binout_t;
```

Fig. 9: Binary output structure.

The ASCII trace-line can consume up to 312 bytes although in our experience it rarely consumes over 128 bytes, nevertheless the traces are large and often impractical to observe with a standard text editor. The overhead is large due to writing out function and variable names which can be considerable even for small applications. The binary format is significantly smaller and consumes just 20 bytes. Figure 9 shows the compacted structure. The key difference is that the binary format stores function and variable names and tags them with an id for later look-up. Moreover, integer values are at most 8 bytes large. The binary output consumes slightly more run-time memory, but has significant disk space savings.

We previously mentioned that enabling tracing negatively impacts overall performance and therefore ultimately determines the tool's upper limit. The general rule is that overall performance is I/O bound. Clever data-compressing techniques can improve performance; however, at this stage the tool only employs a custom binary output. We compared the tool's performance as well as overall and average trace sizes when running on several nodes/PE combinations.

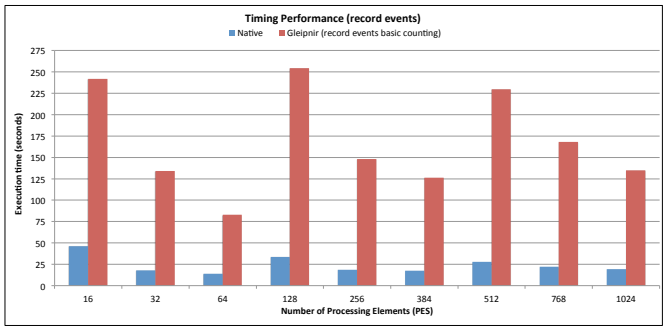


Fig. 10: Timing with basic event counting.

Figure 10 shows Gleipnir's performance compared to the native run when basic event counting is enabled. Note that this does not show the entire application. Using *perftools-lite* we determined application functions which contribute a significant amount to execution time, and we enabled basic event counting around this function call. From the figure we can observed that the average overhead is approximately $5\times$ compared to native performance. This is about $2\times$ slower than Nullgrind or Gleipnir's *fast-forward* mode.

```
for(outer){
  GL_GLOBAL_START_INSTR;
    i = ilist[ii];
    qtmp = q[i];
    xtmp = x[i][0];
    ytmp = x[i][1];
    ztmp = x[i][2];
    itype = type[i];
    jlist = firstneigh[i];
    jnum = numneigh[i];
  GL_MARK_STR("FAST_FORWARD_ON");
  GL_FAST_FORWARD_ON;
  for(inner){...}
}
```

Fig. 11: An instrumented code snippet.

We will now describe the disk usage for ASCII and binary output. We start with a small example, and compare both outputs over varying number of processing elements. The main computational component as identified by *perftools-lite*

is the *compute()* method in *pair_lj_charmm_coul_long.cpp*. The method consists of a nested loop that contains the main computational body.

*1) ASCII output:* Suppose that we are tracing a small code section in the outer loop as shown in Figure 11. A single iteration of 23 instructions produces trace data of approximately 70bytes per instruction. This means that a 1k loop iteration will generate a trace-file of 70kbytes, and in our example a single computational pass is about 4k iterations. Even this simple example generated ≈6.7MBs per file, and ≈100MB for a single node with 16 PEs running. While this may not be big overhead in itself tracing the inner loop quickly adds several orders of magnitude trace data. Thus tracing statements in the inner loop produces ≈1.65GB trace-files, and ≈25GB for a single node with 16PEs running.

*2) Binary output:* The Binary output is intended to reduce overall disk-space overhead, and potentially save execution time. The drawback is that enabling a binary output requires additional program logic which adds to execution time.[5] For example consider tagging function and data-structures with an id and storing the id in a structure for later look-up. This implies that we must access te cached values at every trace-line to ensure consistency.

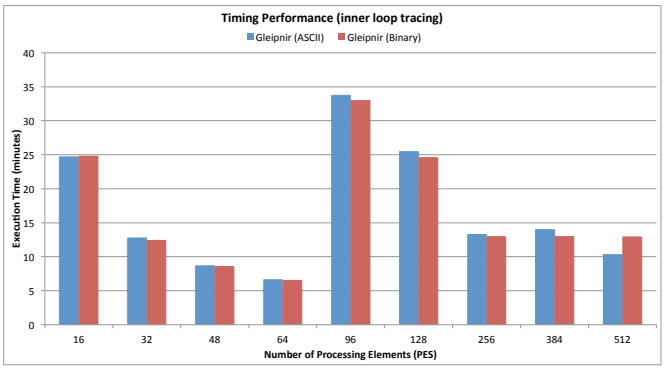


Fig. 12: Timing with inner loop tracing.

Figure 12 shows the overall performance difference between the binary and ASCII output. Similarly to our previous runs we increased the problem size at 96 processing elements. Binary output shows a slight performance improvement compared to the ASCII output.

In Figure 13 we can observe the average trace file size per process. Notice that we increased the problem size after 64 processing elements. The total trace is 25GB for ASCII and 9.5GB for binary output. Per process file size decreases with the increased number of processing elements.

## V. Conclusions

Memory tracing is a valuable tool for performance analysis and we anticipate that such tools will become of great assistance to performance portability planning for future systems. The goal of this paper was to study the scalability of our

[5]The binary mode was developed specifically to address trace sizes at scale, thus most of its functionality is not finished.

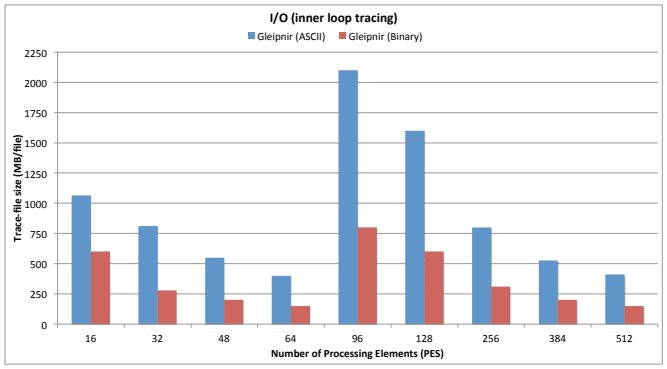


Fig. 13: Average trace-file sizes with inner loop tracing.

Valgrind-based memory tracing technology, Gleipnir, on our Cray XK6 cluster, Titan. We wanted, in particular, to assess the feasibility of realistically tracing parallel applications. We started by providing an introduction to memory tracing technologies and the design of Gleipnir and proceeded with carrying out our investigation in the context of the LAMPPS molecular dynamics simulation code.

There are two facets in the overhead induced by tracing: the slowdown due to the tracing logic and the disk IO that such, fine-grain, tracing may cause. In our experiment, we targeted one of LAMMPS' core force-calculating loops in 16-1024 MPI process settings. Valgrind introduces a 3-fold slowdown, which is in turn amplified to a total of a 100-fold slowdown by Gleipnir. This is perfectly in line with other fine-grain tracing tools. Nonetheless, the application executes to completion with all network IO passing through Valgrind substrate. Working an 30,000-atom problem size, traces averaged a total of 200GB for a single computational step. The average per process trace size difference is within the 6.6% range which is due to the imbalance in the atom distribution. Our experimental switch from the original ASCII format to a binary format yielded 2.6-fold savings, bringing the traces set down to 76GBs. While this reduces the storage footprint notably, the trace entry payload has not affected the execution times – this may as well be due to the number of IO operations remaining unchanged in count. Smaller trace files, however improve the ability to trace larger program regions.

Using Gleipnir for memory tracing parallel applications is a promising technology, which has been shown to work in our Cray setting. We are currently investigating further trace compaction schemes and look forward to better Valgrind support for our system's instruction set.

## Acknowledgements

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. We would like to thank Mike Brim for his valuable help in profiling Valgrind, and Cristian Cira with his help of the LAMMPS code.

## REFERENCES


[1] T. Janjusic, K. M. Kavi, and B. Potter, "International conference on computational science, iccs 2011 gleipnir: A memory analysis tool," *Procedia CS*, vol. 4, pp. 2058–2067, 2011.

[2] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *SIGPLAN Not.*, vol. 42, pp. 89–100, June 2007. [Online]. Available: http://doi.acm.org/10.1145/1273442.1250746

[3] M. D. H. Jan Edler, "Dineroiv trace-driven uniprocessor cache simulator." [Online]. Available: http://www.cs.wisc.edu/~markhill/DineroIV

[4] N. Nethercote, R. Walsh, and J. Fitzhardinge, "Building workload characterization tools with valgrind," Invited tutorial, Los Alamitos, CA, USA, p. 2, October 2006. [Online]. Available: http://valgrind.org/docs/iiswc2006.pdf

[5] J. Weidendorfer, M. Kowarschik, and C. Trinitis, "A tool suite for simulation based analysis of memory access behavior," in *In Proceedings of International Conference on Computational Science*. Springer, 2004, pp. 440–447.

[6] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa, and R. K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *In Programming Language Design and Implementation*. ACM Press, 2005, pp. 190–200.

[7] A. Srivastava and A. Eustace, "Atom: A system for building customized program analysis tools." ACM, 1994, pp. 196–205.

[8] B. Buck and J. K. Hollingsworth, "An api for runtime code patching," *Int. J. High Perform. Comput. Appl.*, vol. 14, pp. 317–329, November 2000. [Online]. Available: http://dl.acm.org/citation.cfm?id=1080622.1080630

[9] D. L. Bruening, "Efficient, transparent and comprehensive runtime code manipulation," Tech. Rep., 2004.

[10] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige, "Tainttrace: Efficient flow tracing with dynamic binary rewriting," in *Proceedings of the 11th IEEE Symposium on Computers and Communications*, ser. ISCC '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 749–754. [Online]. Available: http://dx.doi.org/10.1109/ISCC.2006.158

[11] Q. Zhao, J. E. Sim, W. fai Wong, and L. Rudolph, "DEP: detailed execution profile," in *In PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*. ACM Press, 2006, pp. 154–163.

[12] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "Papi: A portable interface to hardware performance counters," in *In Proceedings of the Department of Defense HPCMP Users Group Conference*, 1999, pp. 7–10.

[13] S. S. Shende and A. D. Malony, "The Tau Parallel Performance System," *Int. J. High Perform. Comput. Appl.*, vol. 20, pp. 287–311, May 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1125980.1125982

[14] N. Tallent, J. Mellor-Crummey, L. Adhianto, M. Fagan, and M. Krentel, "HPCToolkit: performance tools for scientific computing," *Journal of Physics: Conference Series*, vol. 125, no. 1, p. 012088, 2008. [Online]. Available: http://stacks.iop.org/1742-6596/125/i=1/a=012088

[15] M. Schulz, J. Galarowicz, and W. Hachfeld, "Open SpeedShop: open source performance analysis for Linux clusters," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, ser. SC '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1188455.1188470

[16] S. Plimpton, "Fast Parallel Algorithms for Short-Range Molecular Dynamics," *Journal of Computational Physics*, vol. 117, pp. 1–19, 1995. [Online]. Available: {http://http://lammps.sandia.gov}