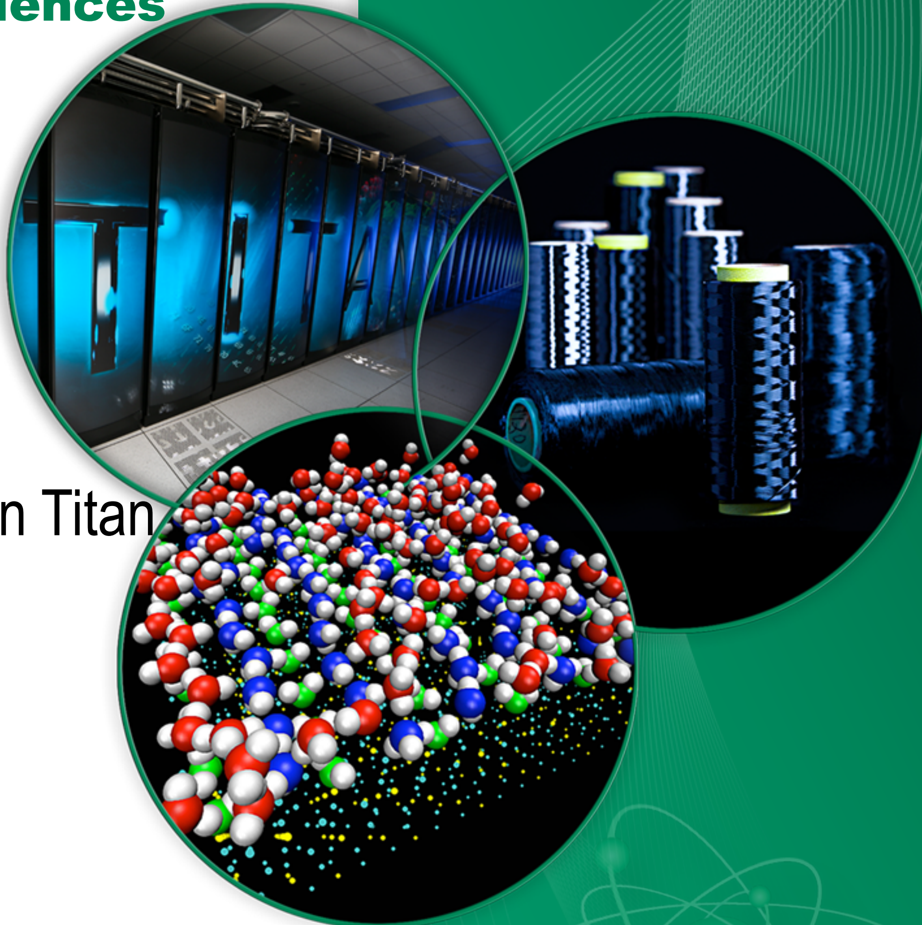
Oak Ridge National Laboratory
Computing and Computational Sciences



Scalability Analysis of Gleipnir:

A Memory Tracing and Profiling Tool, on Titan

Presented by:
Tomislav Janjusic (Tommy)

Introduction

- 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.
- 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.

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.

Introduction

- 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.

- Various frameworks enable the development of fine-grained instruction and memory tracing.
- The general rule is that exposing a greater detail implies a greater "significant" performance overhead, and more importantly recording this detail incurs additional space and time overhead.

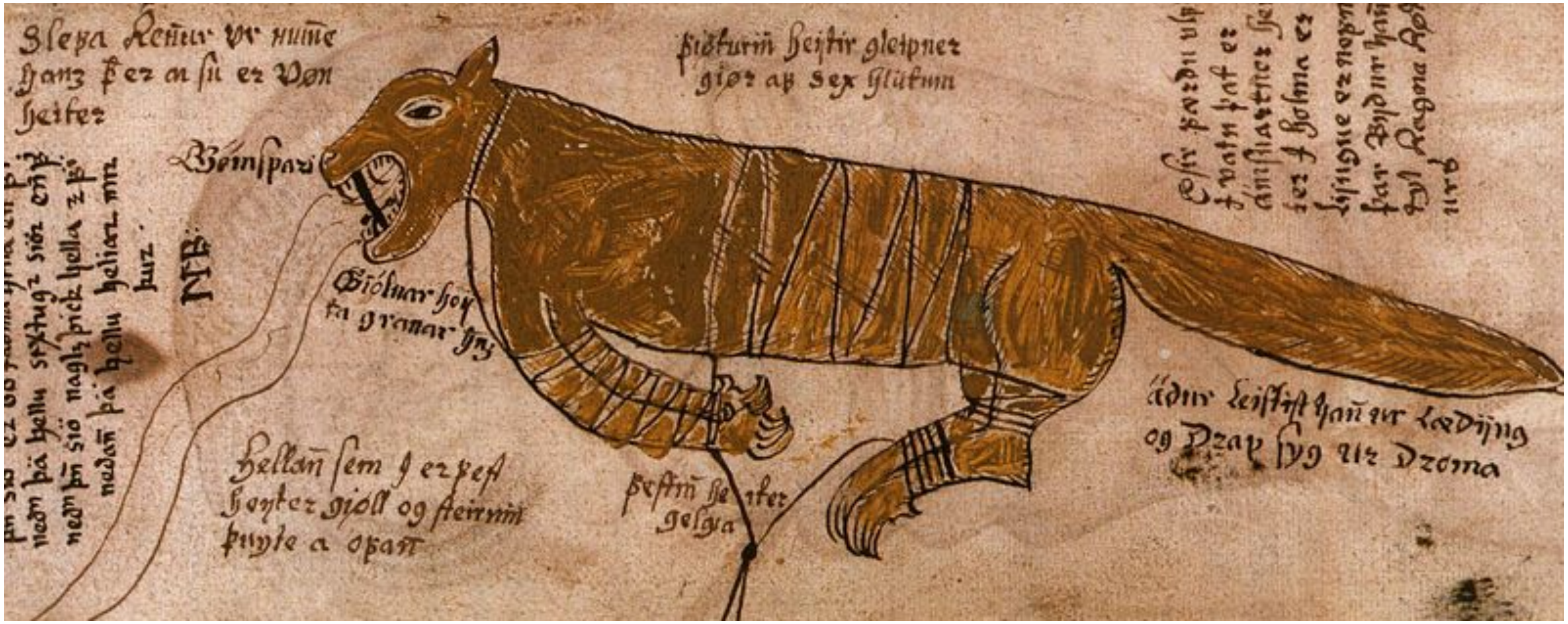
What's in a name?

“Valgrind is the name of the main entrance to Valhalla (the Hall of the Chosen Slain in Asgard). Over this entrance there resides a wolf and over it there is the head of a boar and on it perches a huge eagle, whose eyes can see to the far regions of the nine worlds. Only those judged worthy by the guardians are allowed to pass through Valgrind. All others are refused entrance.
It's not short for "value grinder", although that's not a bad guess.“

– www.valgrind.org

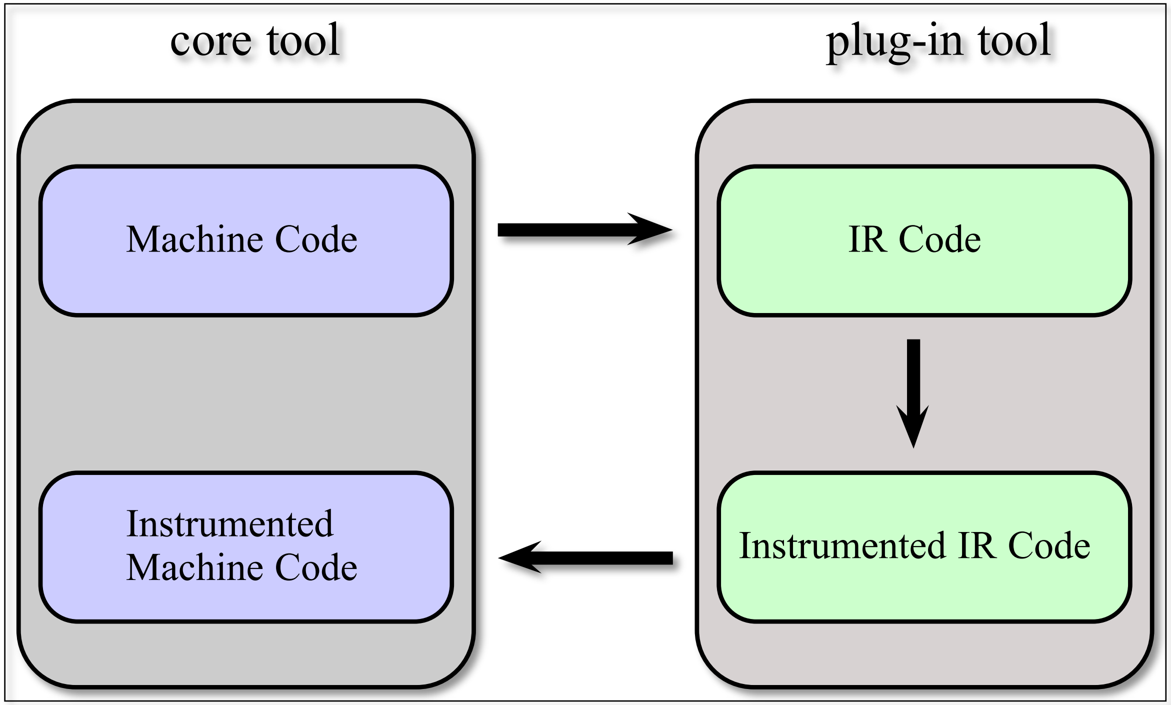
What’s in a name?

“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. The goal of Gleipnir is to help programmers to restrain the memory hogs in applications.“



Gleipnir Overview (Valgrind's IR)

- 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.



Gleipnir Overview (Valgrind's IR)

- 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.

```
static
IRSB* nl_instrument ( VgCallbackClosure* closure,
                      IRSB* bb,
                      VexGuestLayout* layout,
                      VexGuestExtents* vge,
                      VexArchInfo* archinfo_host,
                      IRType gWordTy, IRType hWordTy )
{
    return bb;
}
```

Gleipnir Overview (Tracing Instructions)

- Gleipnir operates on instruction events:
  - instruction read (Ir), data read (Dr), data write (Dw), or data modify (Dm).
- Incoming SB is parsed and instrumented with *dirty helper calls.*
  - The helper function can record any number of events
  - Most are related to instruction's address look-up, debug information annotation, and memory instructions counting/recording.
- Instruction to source code structure/variable mapping:
  - *Local data:* If an application's binary image retained the debug information use Valgrind's debug parser.
  - *Dynamic and Global data:* wrap or replace *malloc()* calls.
    - Wrapping: malloc() call is intercepted, executed, and recorded
    - Replacing: malloc() call is intercepted and redirected. (custom allocators)

Gleipnir Overview (Trace Examples)

```
#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;
}
```

```
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
```

Gleipnir Overview (Trace Examples)

```
#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;
}
```

- Client calls:
  - Manually record dynamic or global objects.
  - Fast forward instrumentation.
  - Trace only interesting sections.

Gleipnir Overview (Trace Examples)

```
#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;
}
```

```
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
```

Gleipnir Overview (Trace Examples)

```c
#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;
}
```

```
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
```

Gleipnir Overview (Trace Examples)

```
#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;
}
```

```
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
```

Gleipnir Overview (Trace Examples)

```
#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;
}
```

```
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
```

Gleipnir Overview (Trace Examples)

```
#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;
}
```

```
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
```

Gleipnir Overview (Trace Examples)

```
#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;
}
```

```
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
```

Gleipnir Overview (Trace Examples)

```
#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;
}
```

```
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
```

Gleipnir Overview (Trace Examples)

```
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
```

Local and Global debug info

vs.

Dynamic debug info

Gleipnir Overview (Trace Examples)

```
#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;
}
```

```
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
```

Gleipnir Overview (Options)

--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

Gleipnir Overview (Options)

- --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

(--fast-forward-on, --trace-state-on) ⟷ Initial trace-state granularity.

Gleipnir Overview (Options)

--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

Control debug and malloc() options.

Gleipnir Overview (Options)

--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

Hints to SB loop about client. (refers to --multi-process and --multi-threaded)

Gleipnir Overview (Options)

--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

Trace physical addresses.

Gleipnir Overview (Options)

```
--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
```

Additional trace information. (applies to --trace-instructions, --trace-malloc-calls, --trace-values)

Gleipnir Overview (Options)

```
--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
```

Profile flush, and naming options.

Gleipnir Overview (Client Interface Calls)

- *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*

Gleipnir Overview (Client Interface Calls)

- *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*

→ Control instrumentation. (applies to the first six calls)

Gleipnir Overview (Client Interface Calls)

- *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*

Client trace hints.

Gleipnir Overview (Client Interface Calls)

- *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*

Manage dynamic/global blocks.

Gleipnir Overview (Client Interface Calls)

- *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*

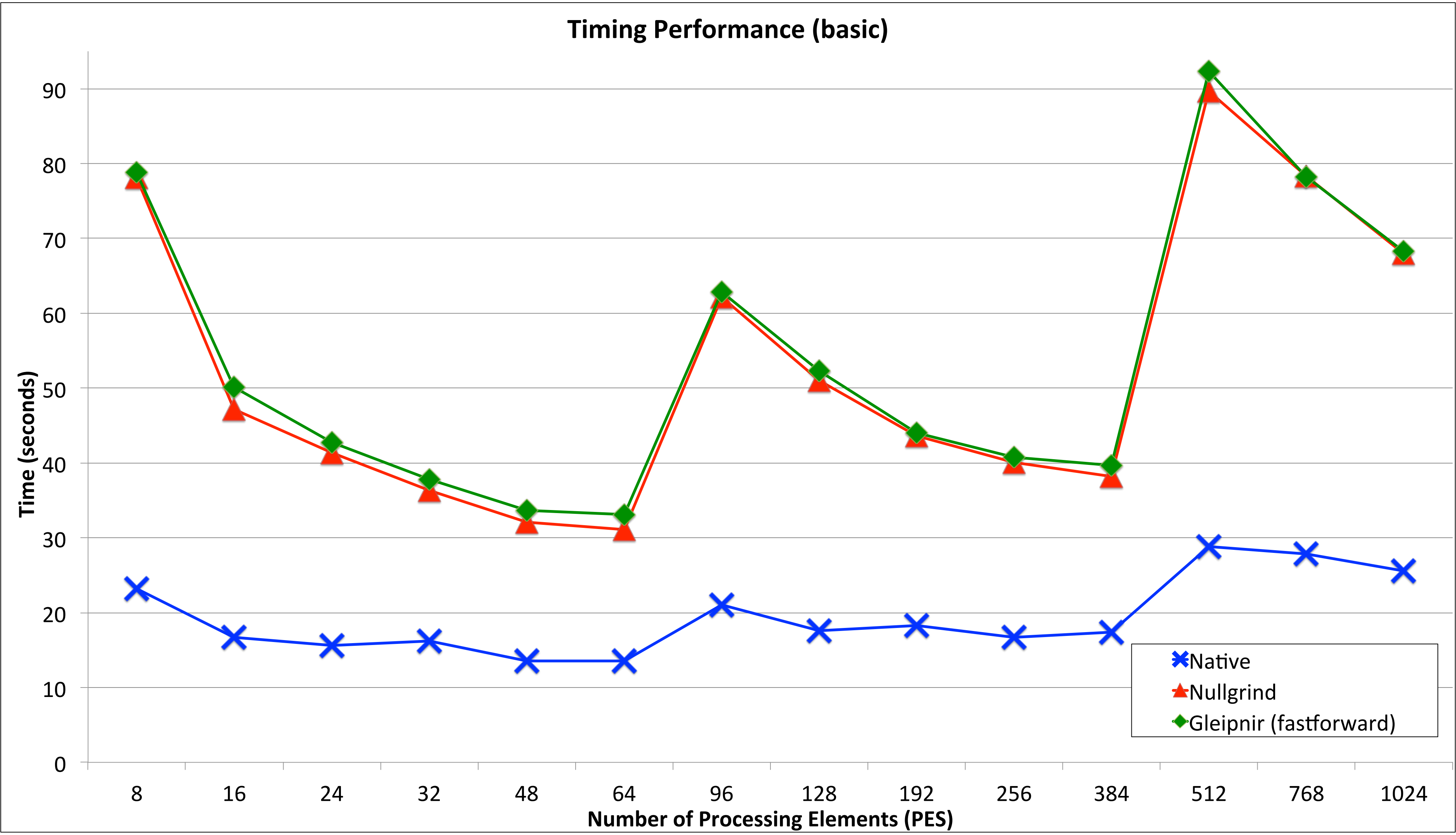
Client user message hints.

Scalability & Performance

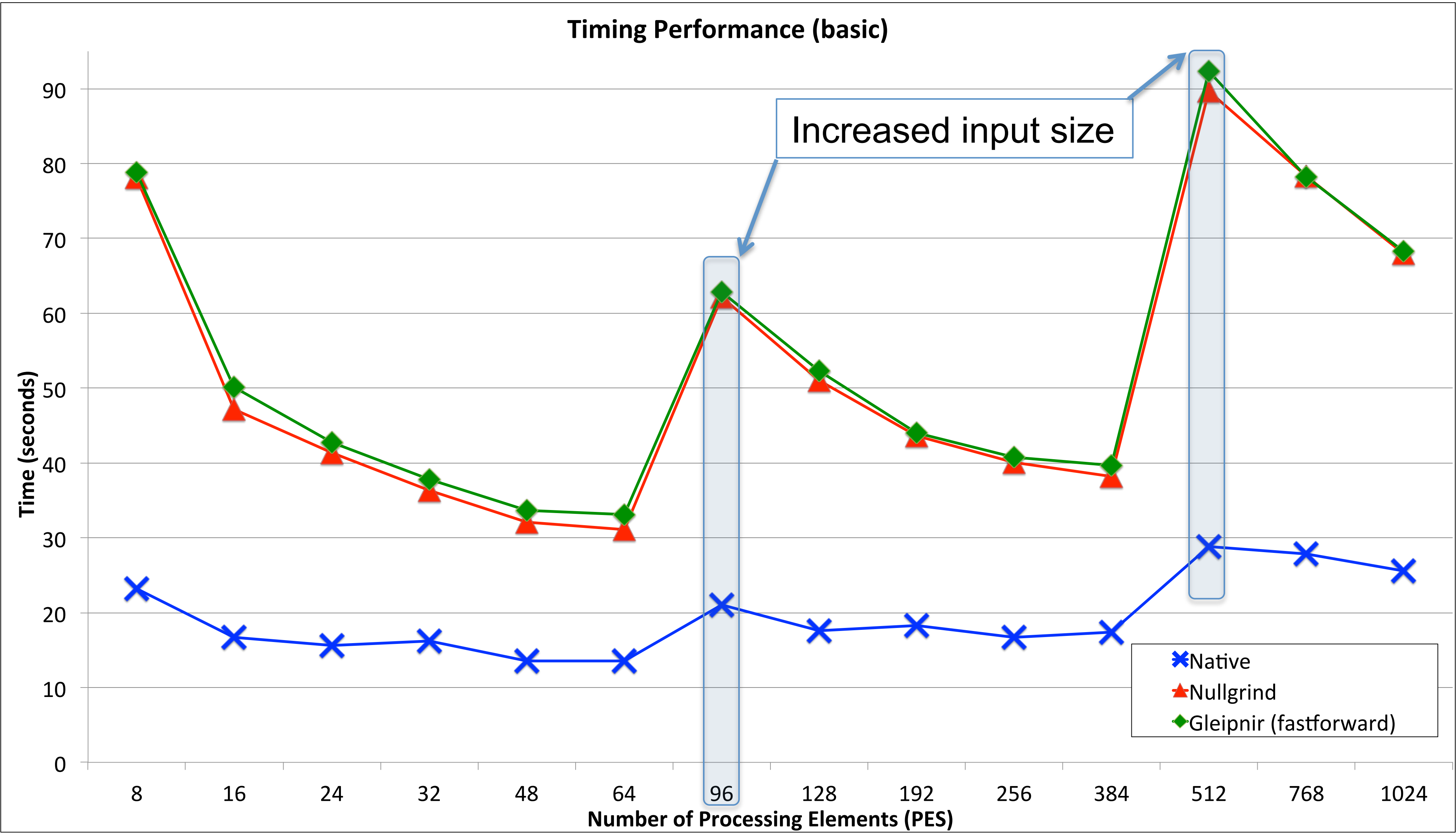
- We conducted our scalability and performance analysis using ORNL's Cray XK-7 Titan system.
- We chose the LAMMPS[16], Large-scale Atomic/Molecular Massively Parallel Simulator, application as our benchmark because of its ease of deployment and scalability.
  - LAMMPS is a classical molecular dynamics 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.

Scalability & Performance

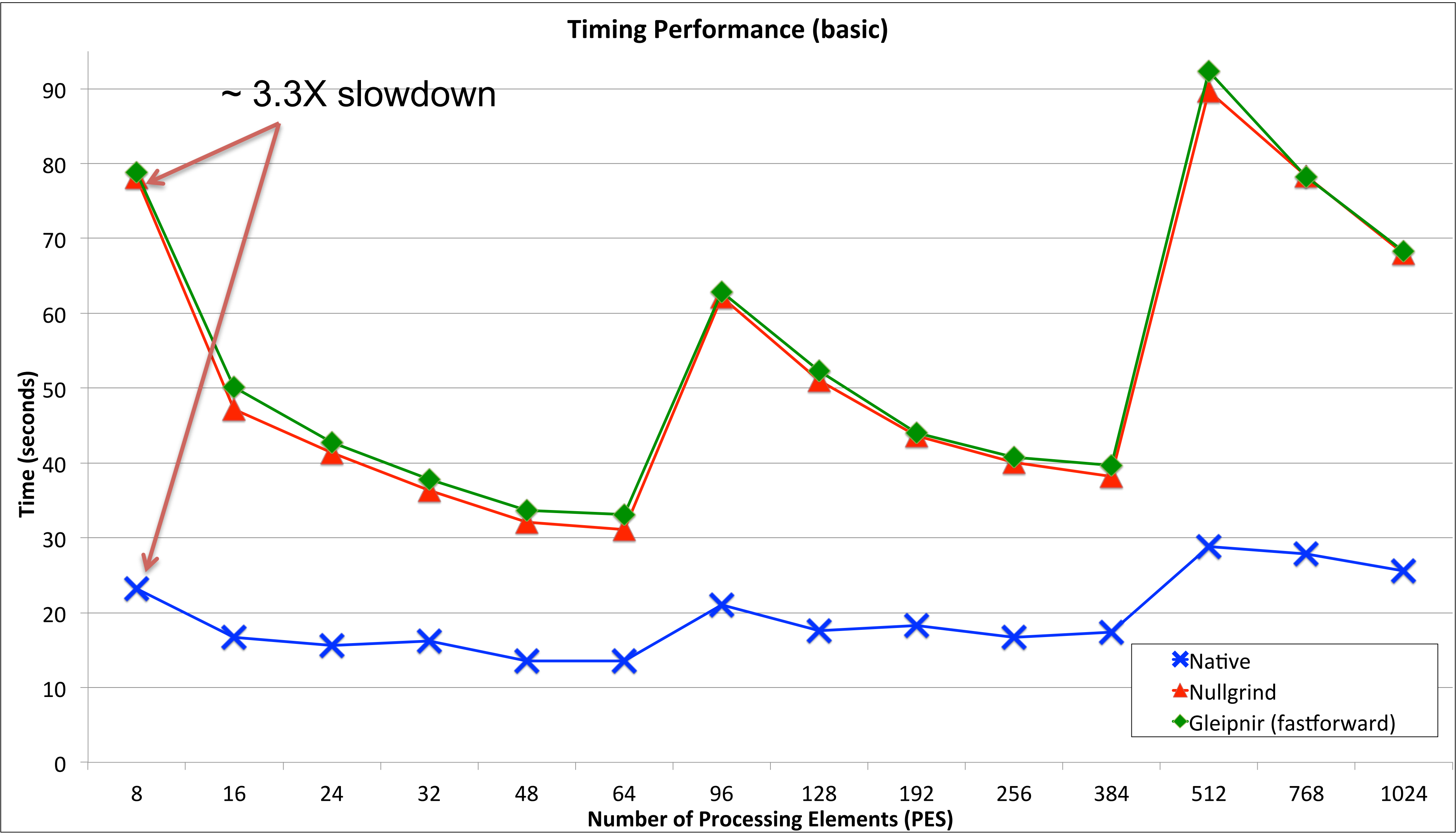
Scalability & Performance (Basic Timing)



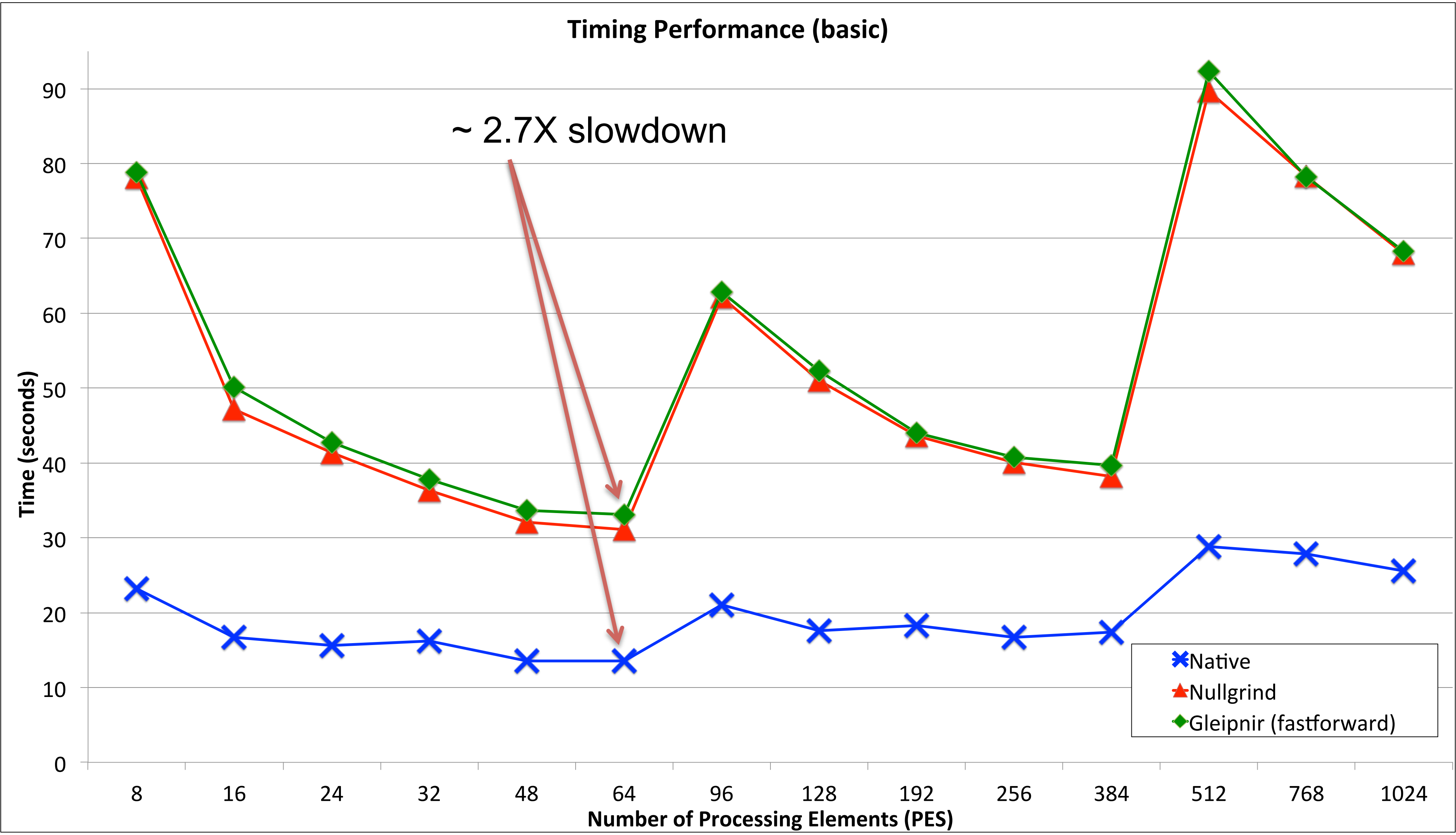
Scalability & Performance (Basic Timing)



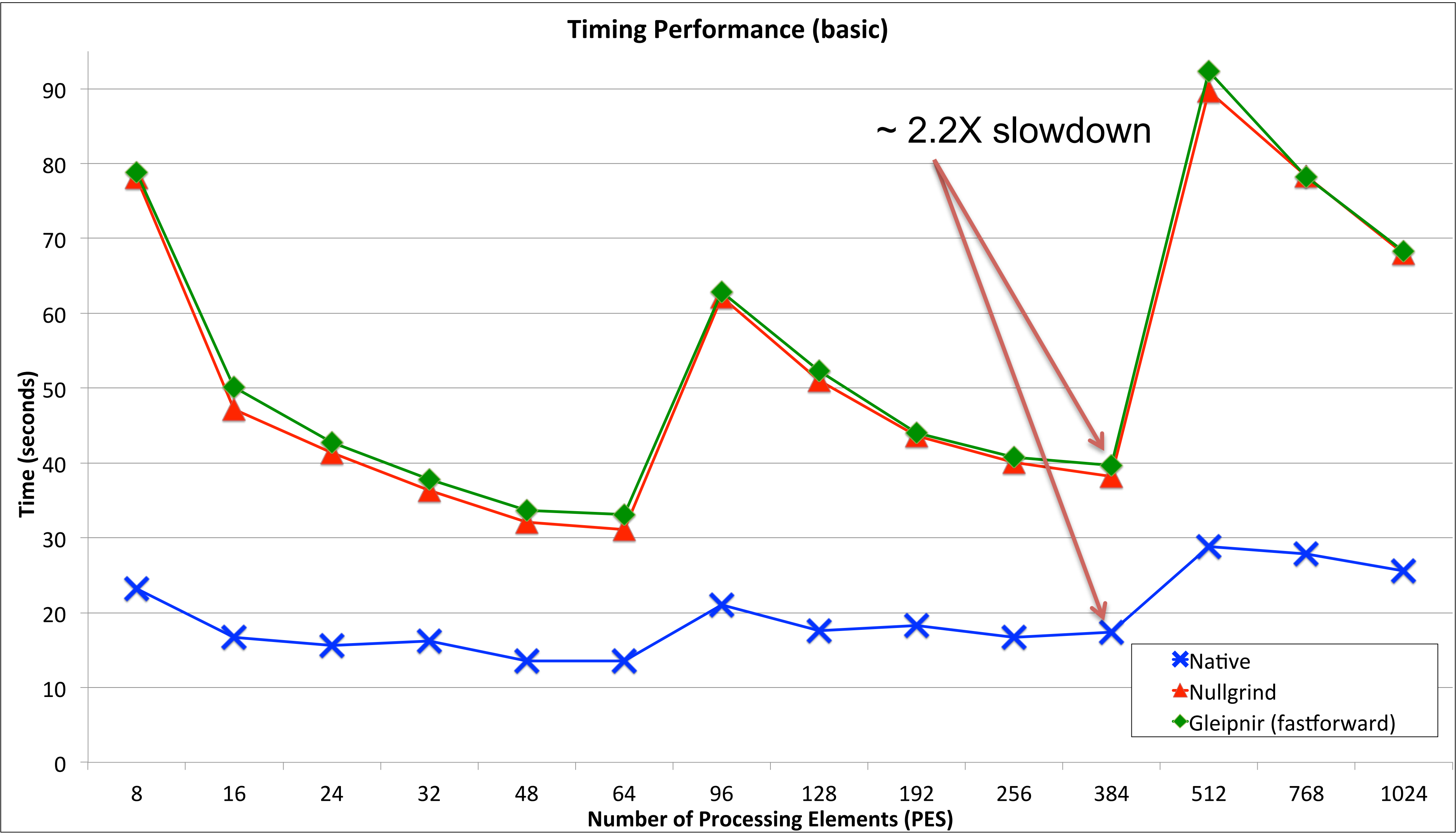
Scalability & Performance (Basic Timing)



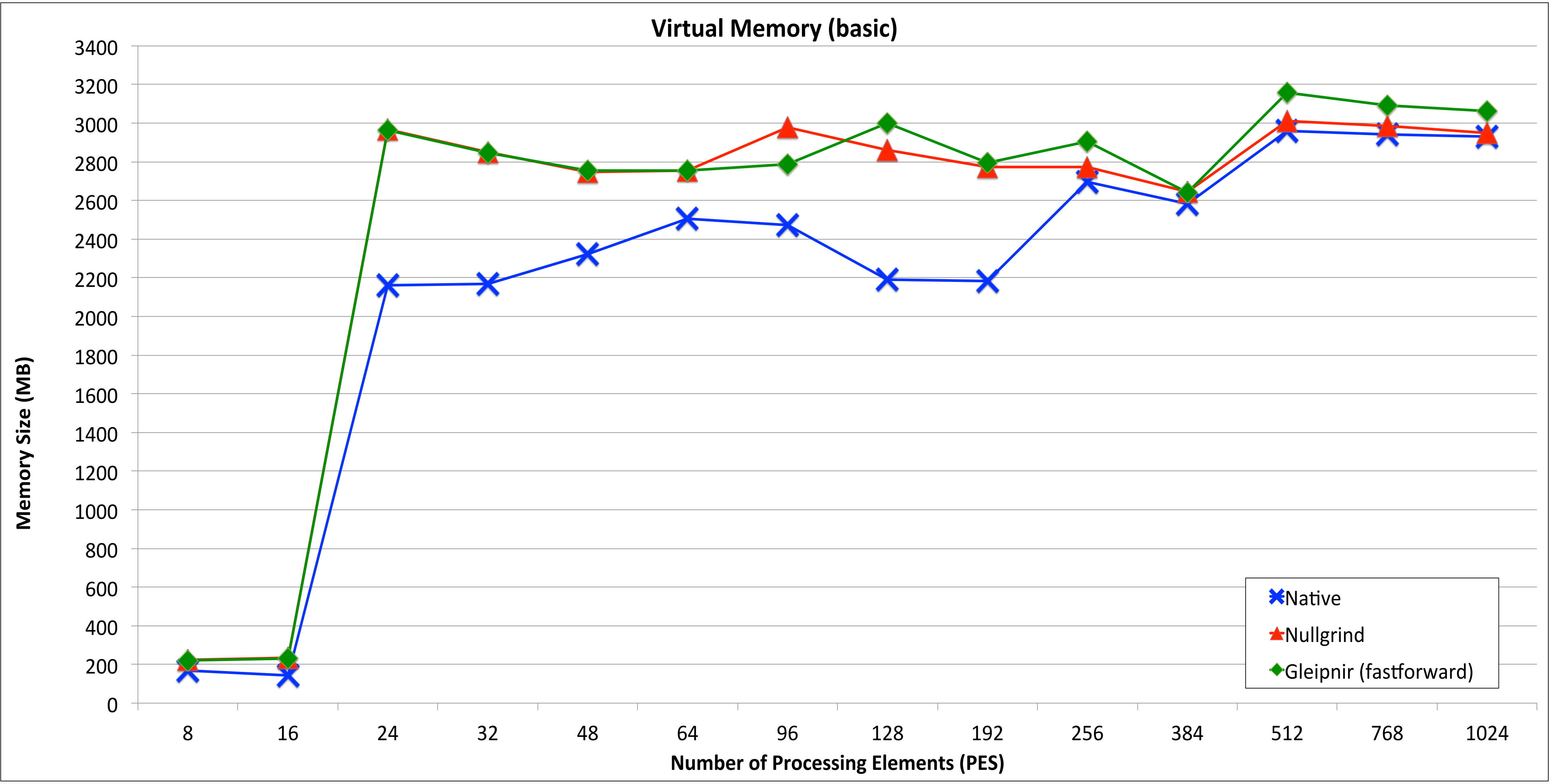
Scalability & Performance (Basic Timing)



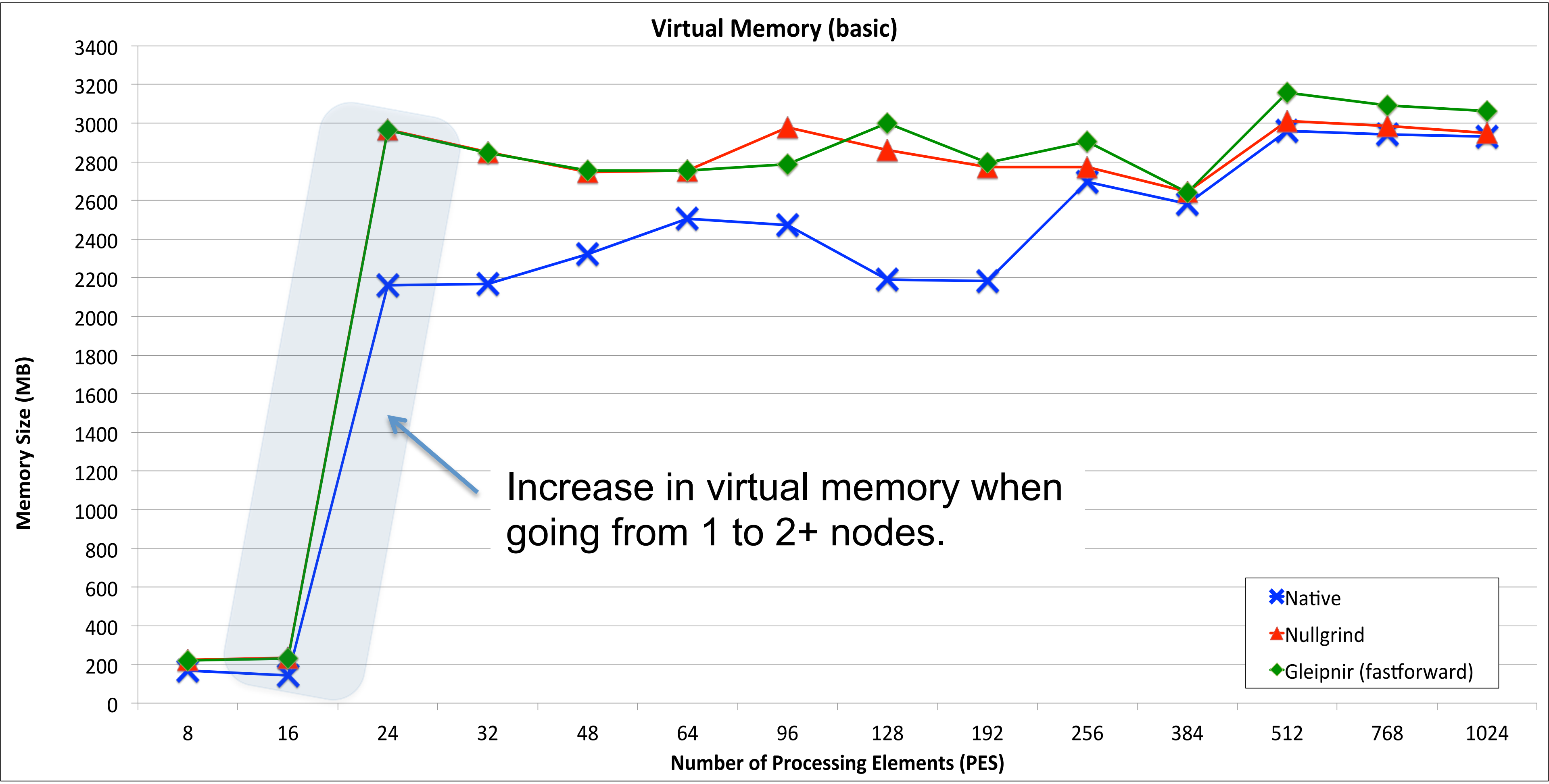
Scalability & Performance (Basic Timing)



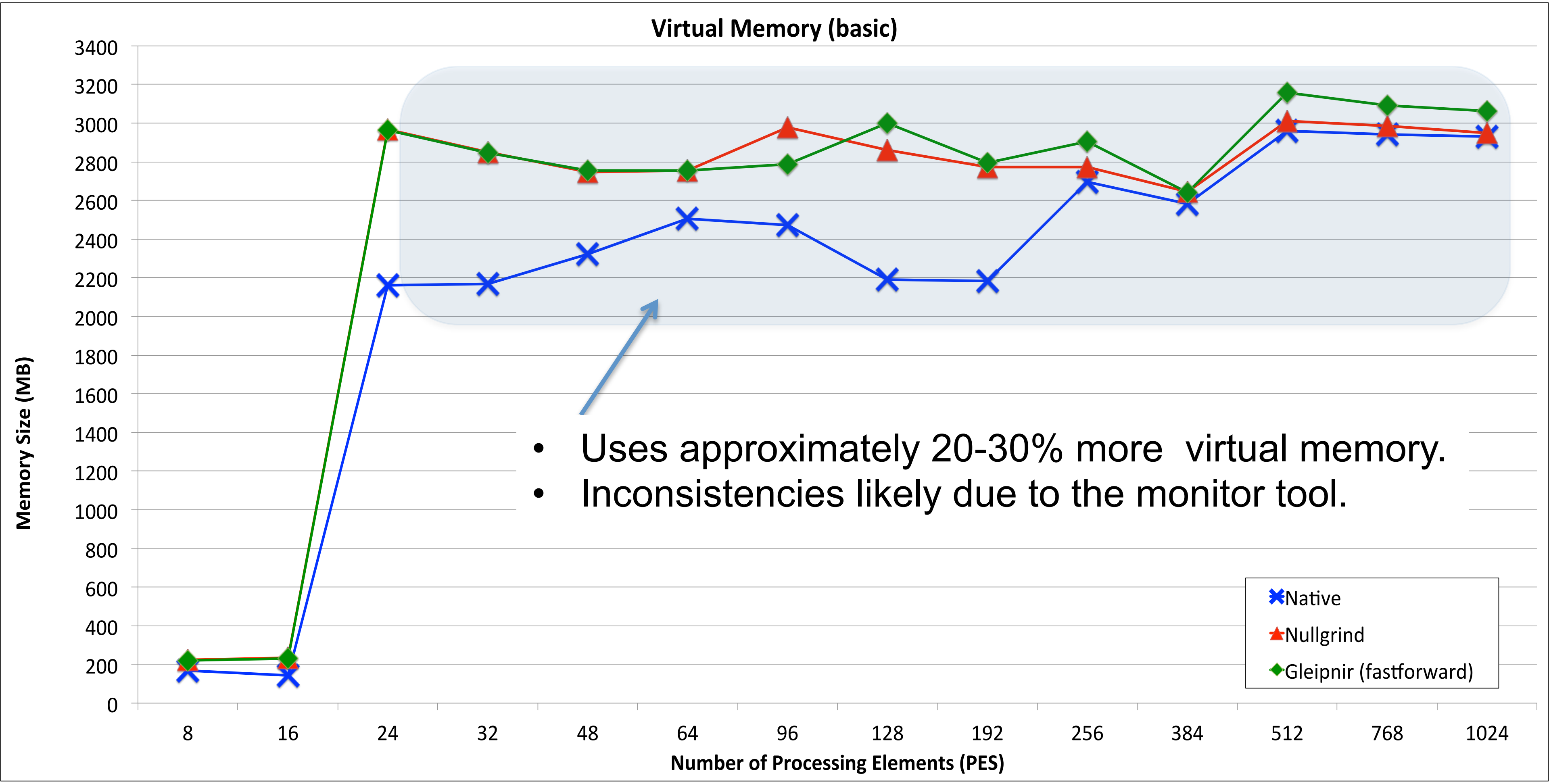
Scalability & Performance (Basic Virtual Memory)



Scalability & Performance (Basic Virtual Memory)

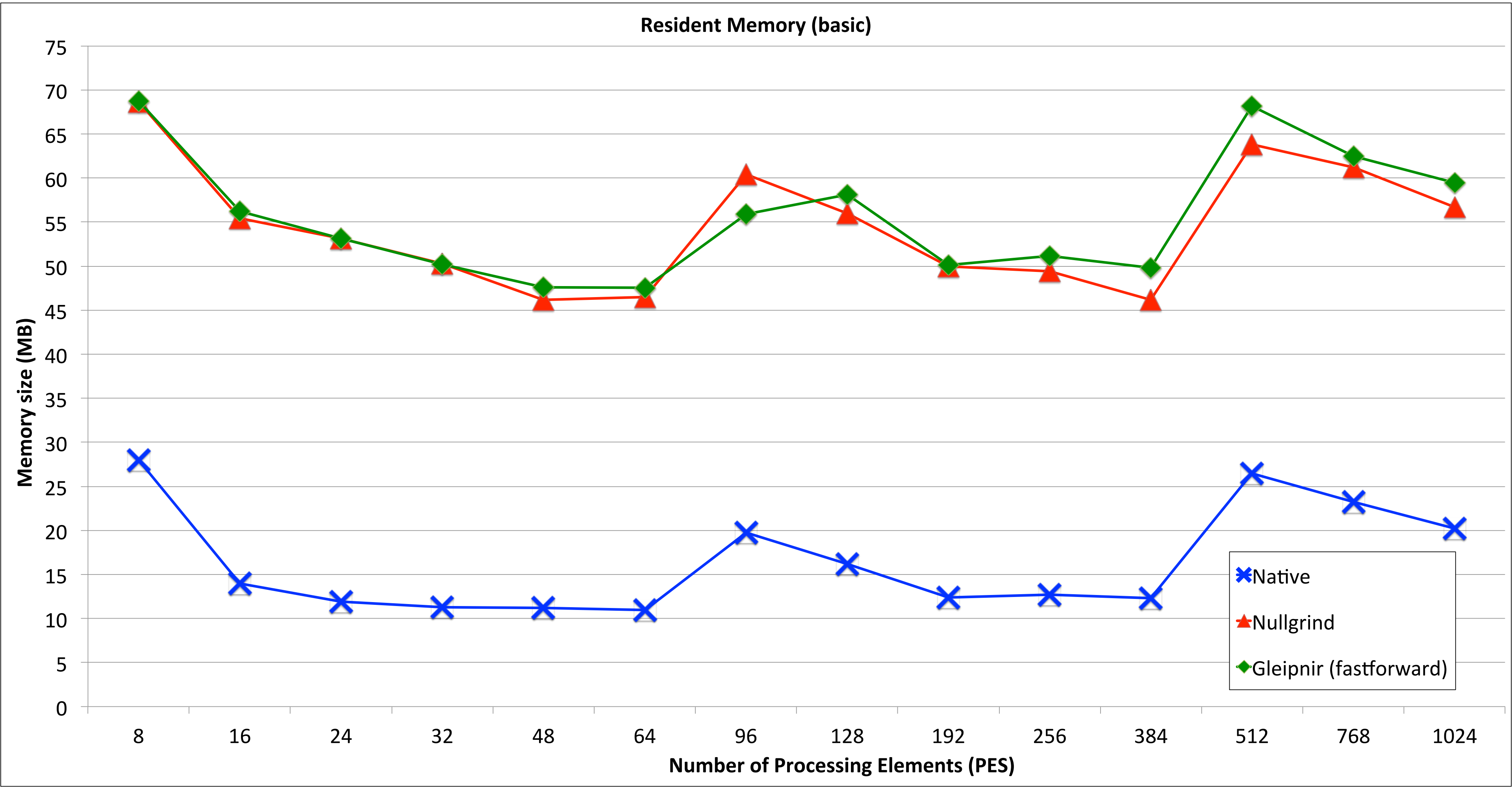


Scalability & Performance (Basic Virtual Memory)

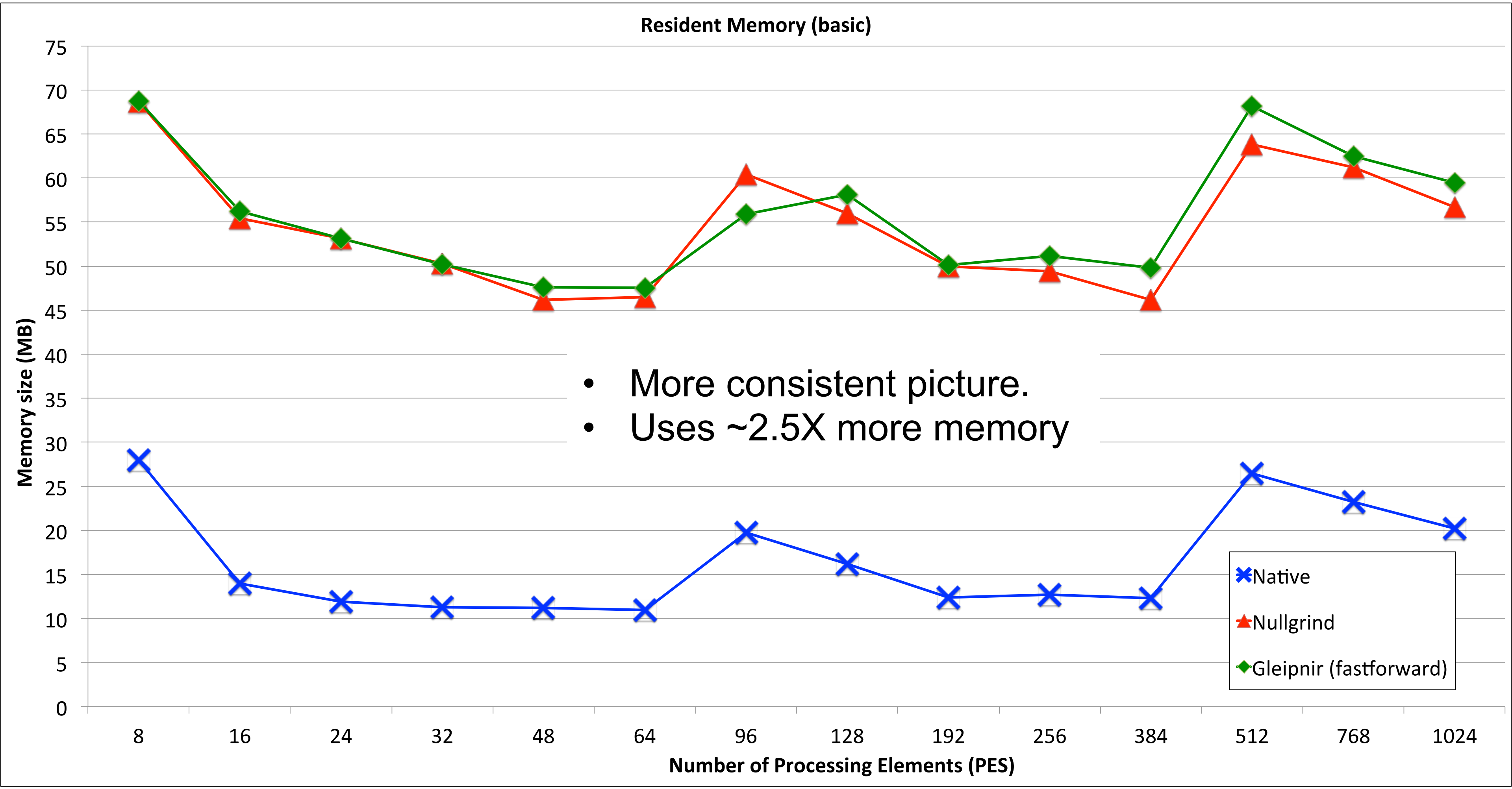


- Uses approximately 20-30% more virtual memory.
- Inconsistencies likely due to the monitor tool.

Scalability & Performance (Basic Resident Memory)



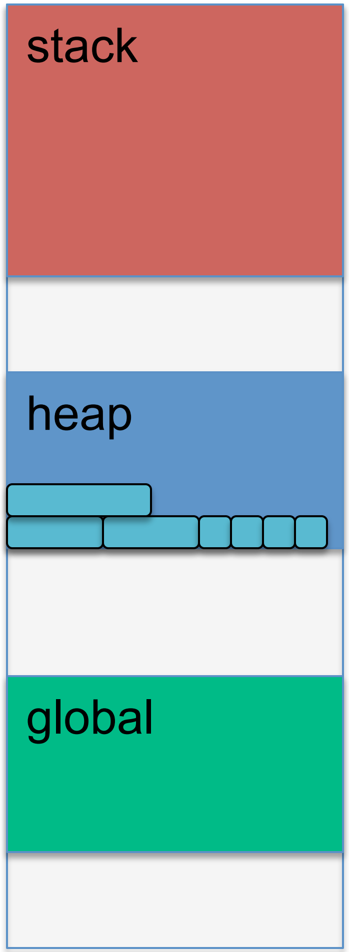
Scalability & Performance (Basic Resident Memory)



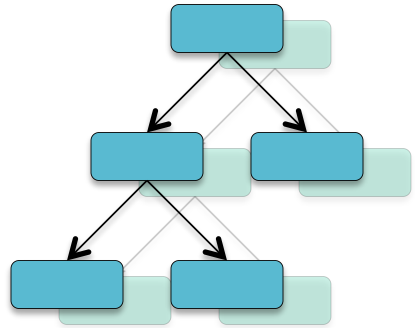
- More consistent picture.
- Uses ~2.5X more memory

Scalability & Performance (More on memory requirements)

Virtual Address Space



Recorded memory blocks



- Every malloc is intercepted and recorded
- Blocks can be manually marked, updated, or removed
- Every de-allocation will free recorded blocks

Scalability & Performance (More on memory requirements)

```
typedef struct _HB_Chunk
{
  Addr data;                    /* start address */
  HChar struct_name[128];
  SizeT req_szB;                /* size requested */
  SizeT slop_szB;               /* slop size */
  ULong trefs;
  ULong refs;
  ULong enum_name;
} HB_Chunk;
```

Scalability & Performance (More on memory requirements)

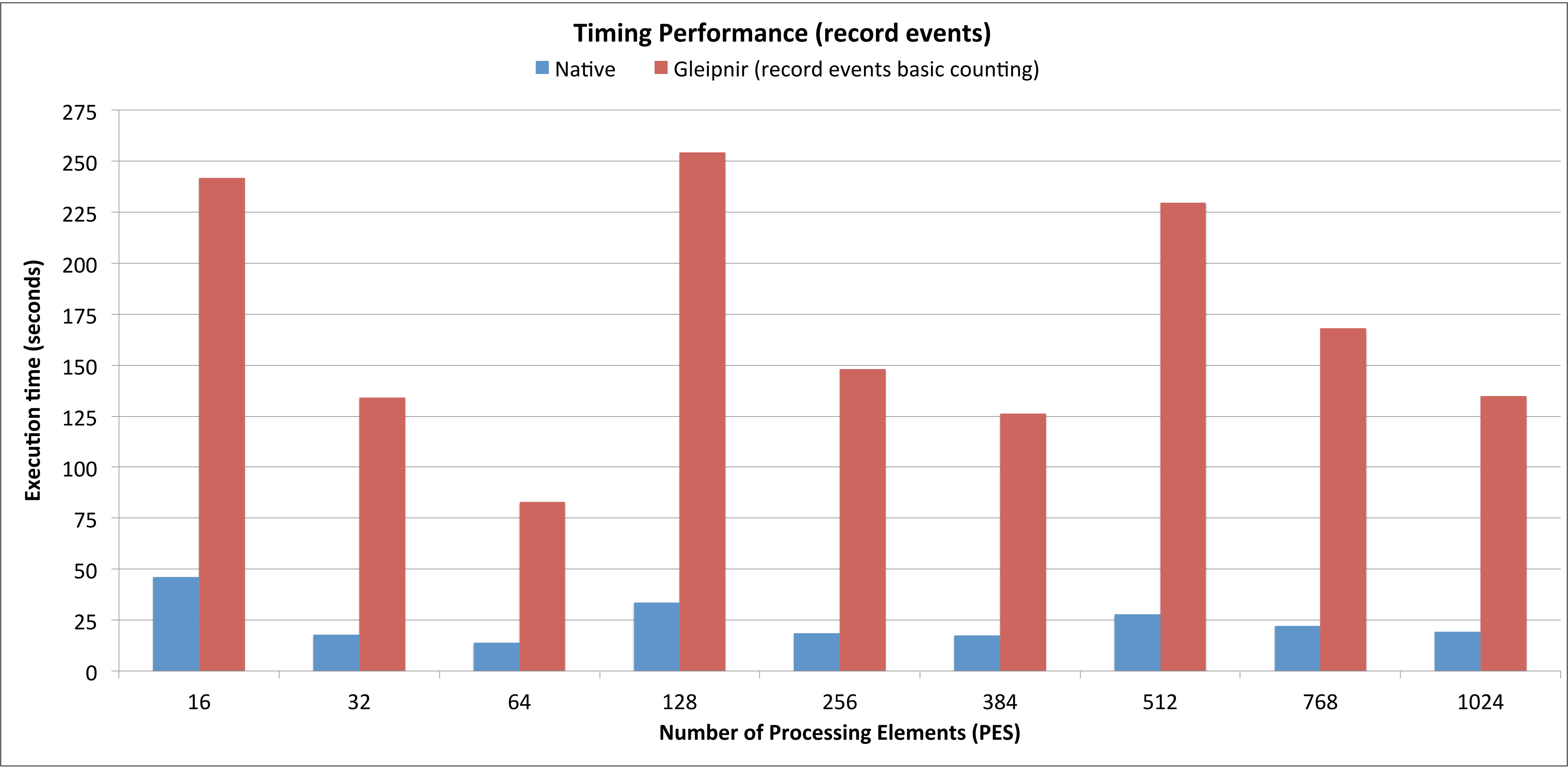
- Gleipnir's primary memory overhead is dictated by the application's memory allocation pattern.

- Coarse allocation pattern will allocate fewer blocks.
- Fine-grained memory allocation pattern will have more allocations thereby increasing the number of memory regions to track.

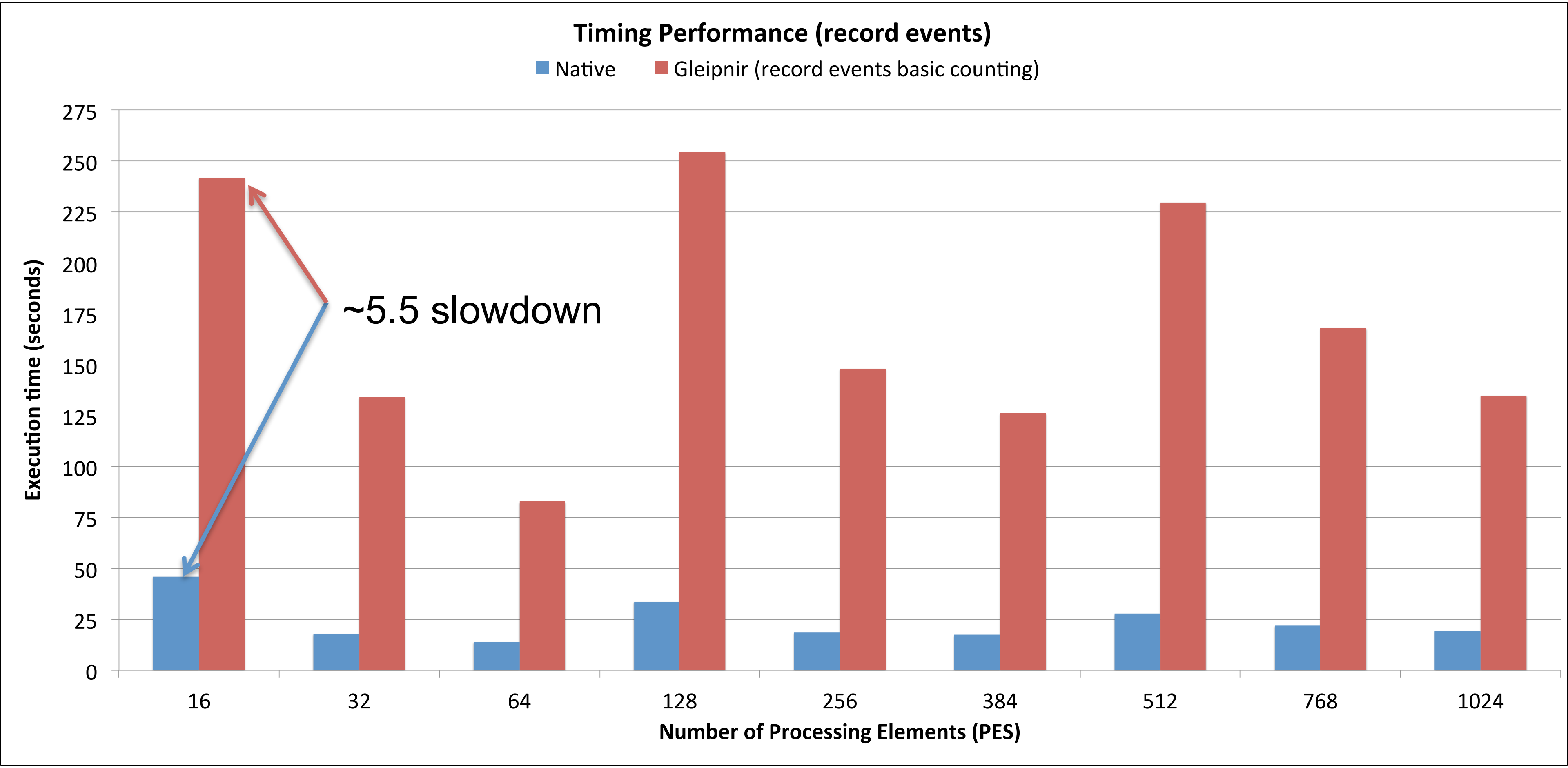
- Valgrind's memory overhead is approximately one extra bit per byte.

- The general rule is that for every allocation Gleipnir will use an additional 168 bytes.

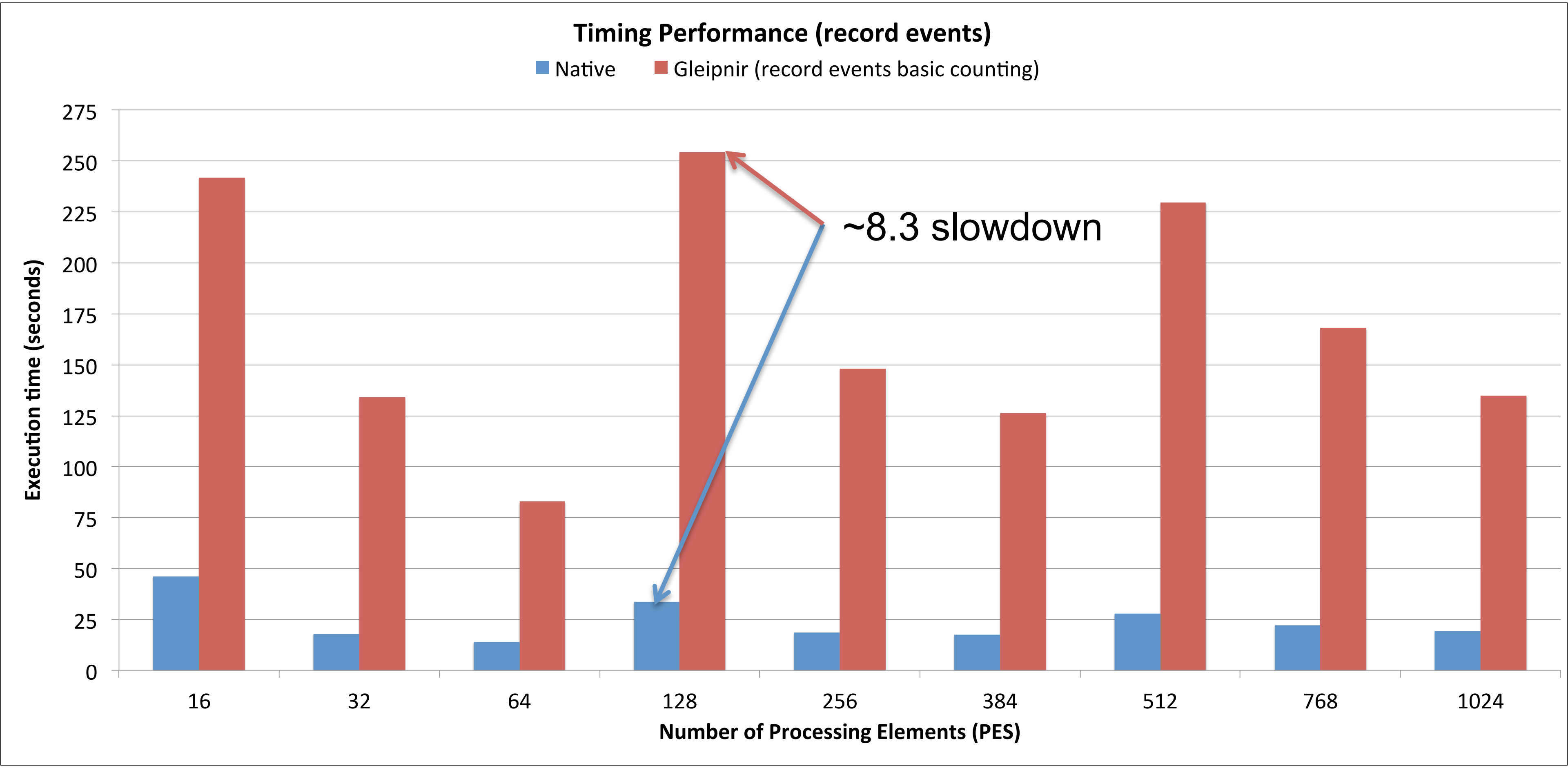
Scalability & Performance (Tracing and I/O)



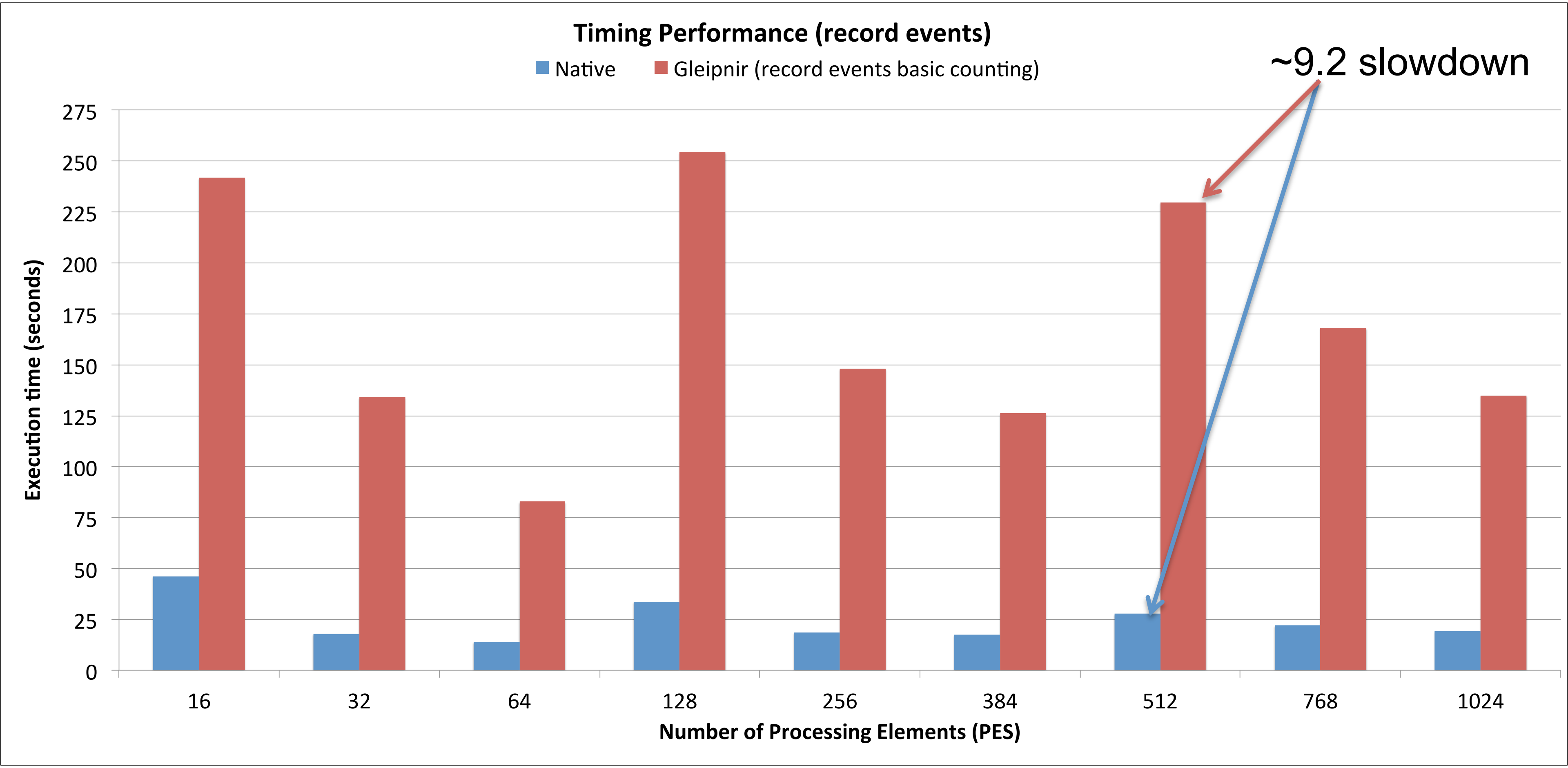
Scalability & Performance (Tracing and I/O)



Scalability & Performance (Tracing and I/O)



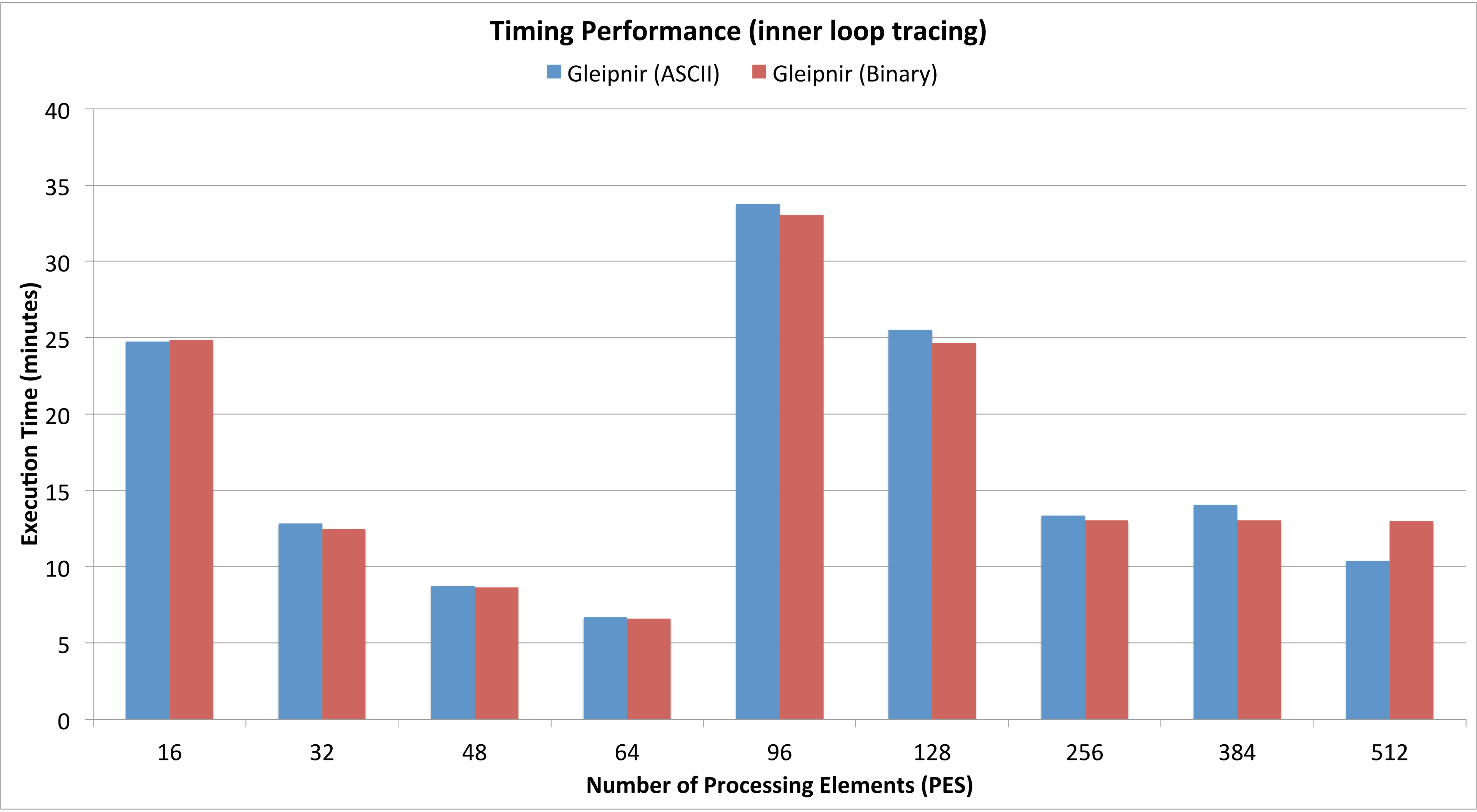
Scalability & Performance (Tracing and I/O)



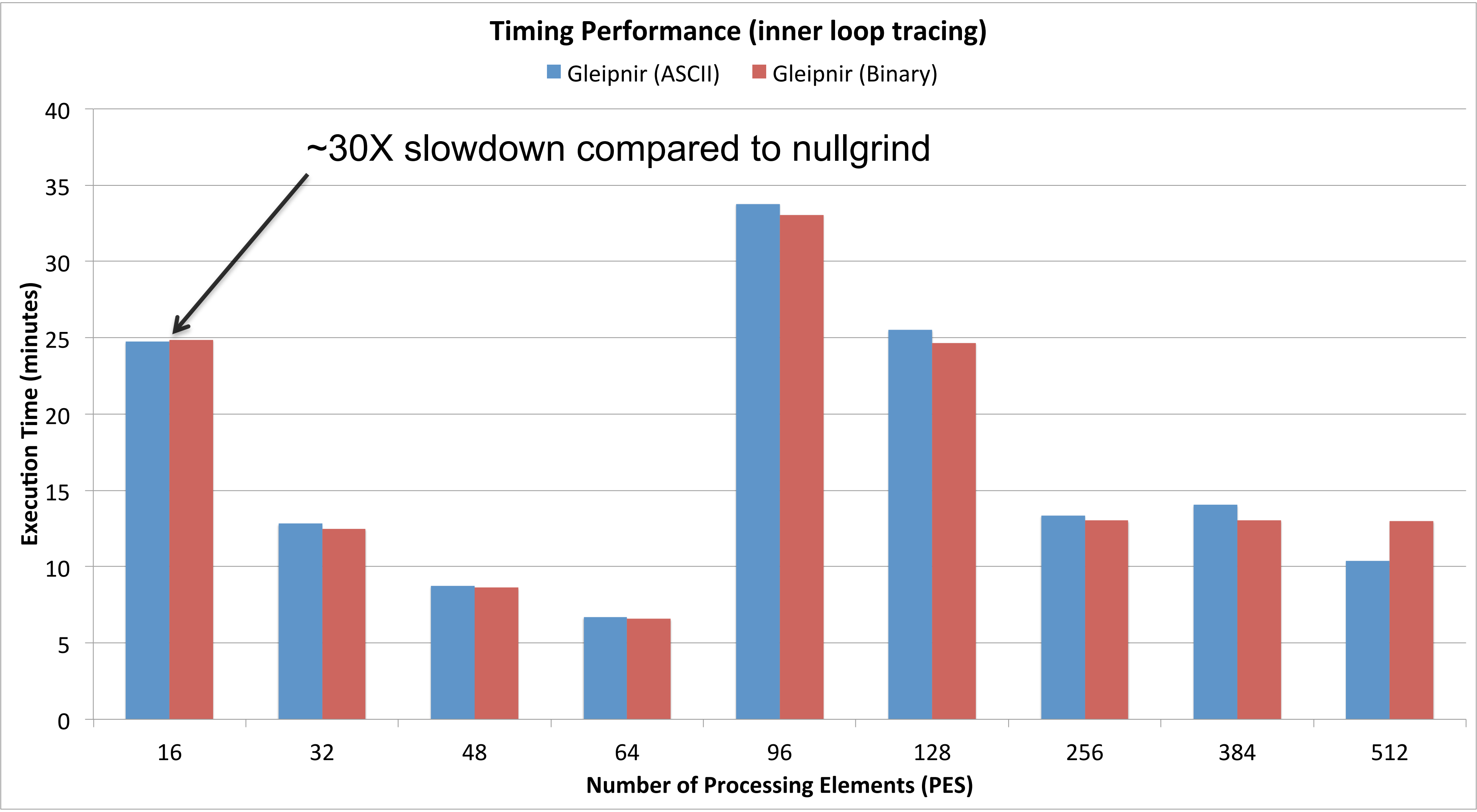
Scalability & Performance (Tracing and I/O)

```
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){...}
}
```

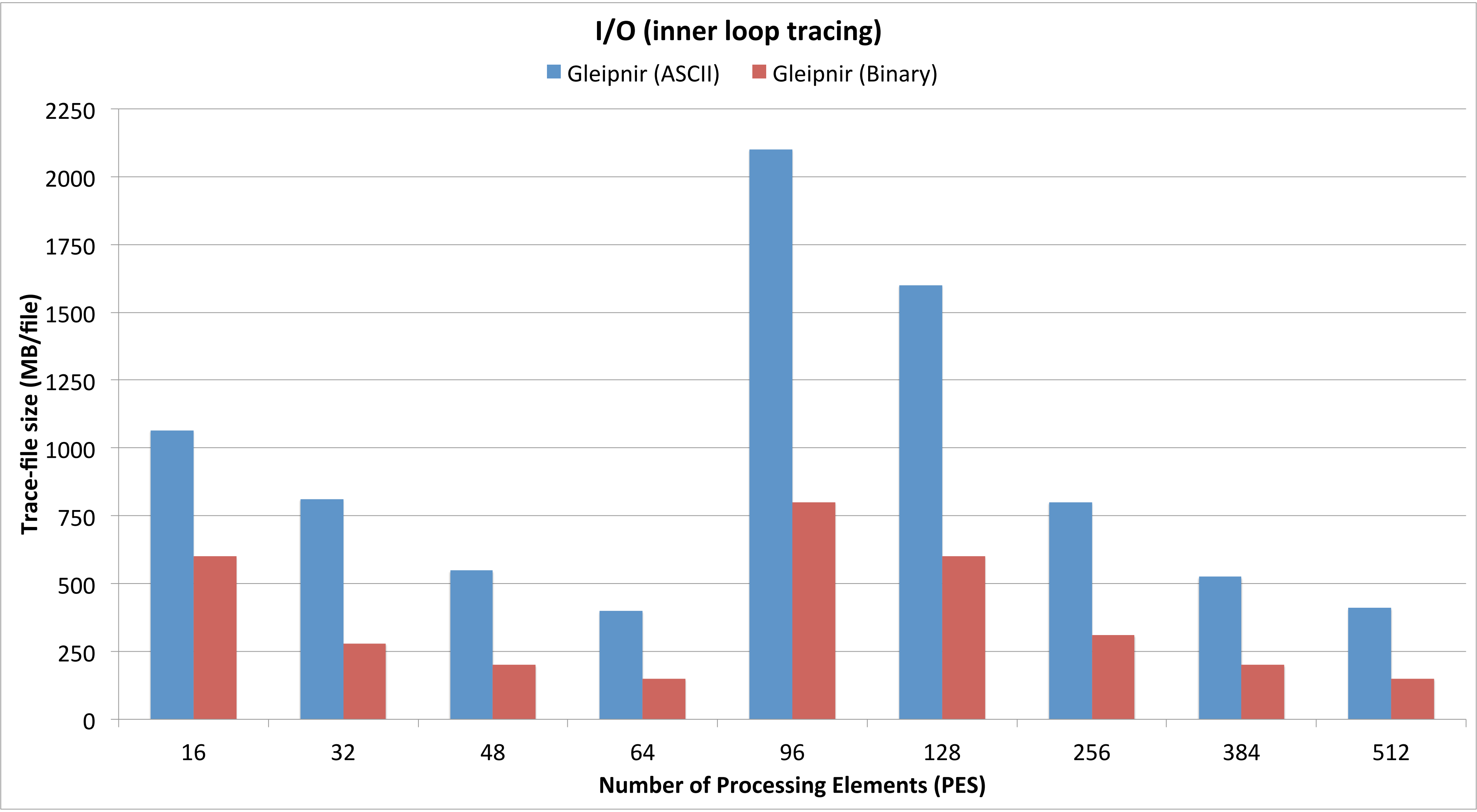
Scalability & Performance (Tracing and I/O)



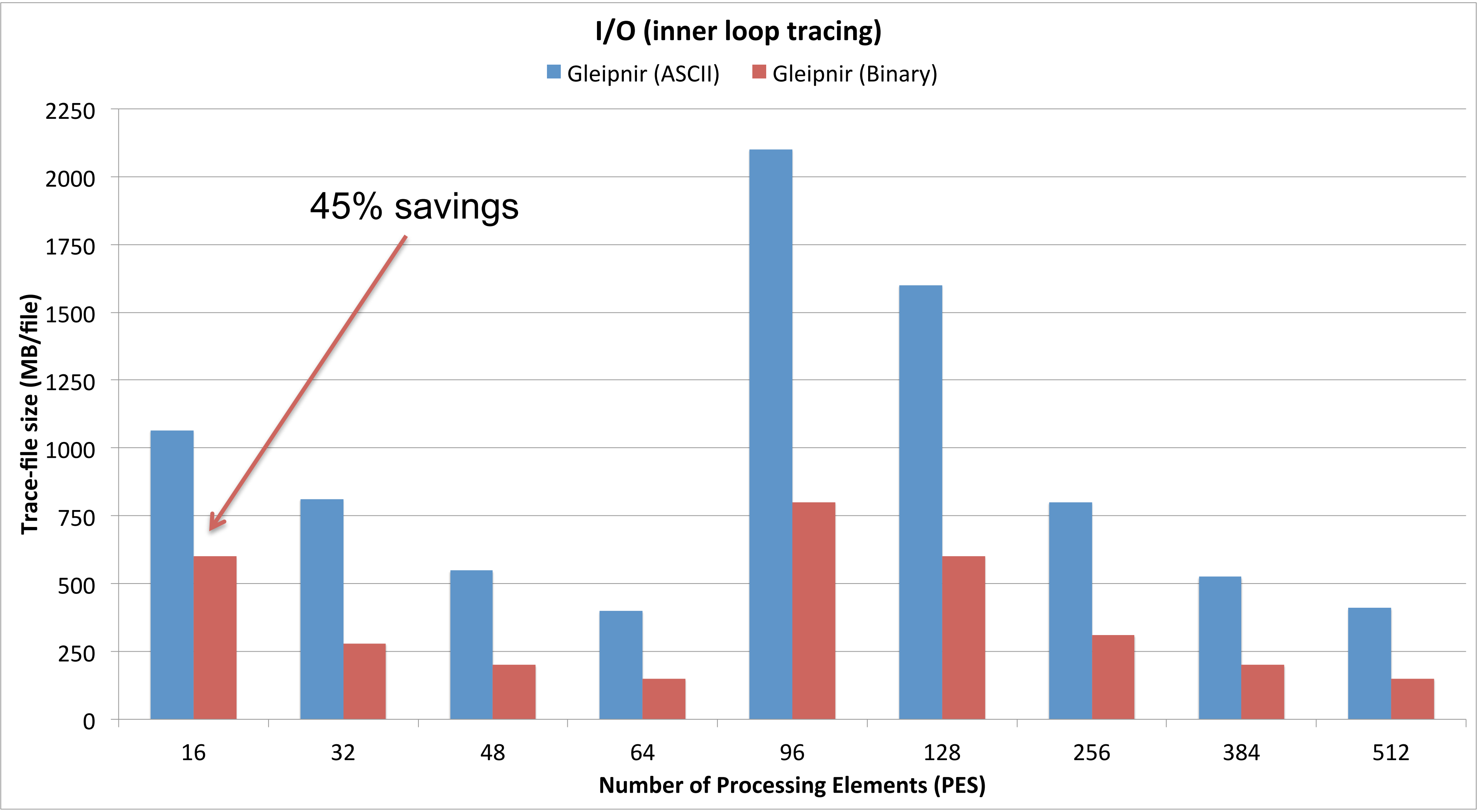
Scalability & Performance (Tracing and I/O)



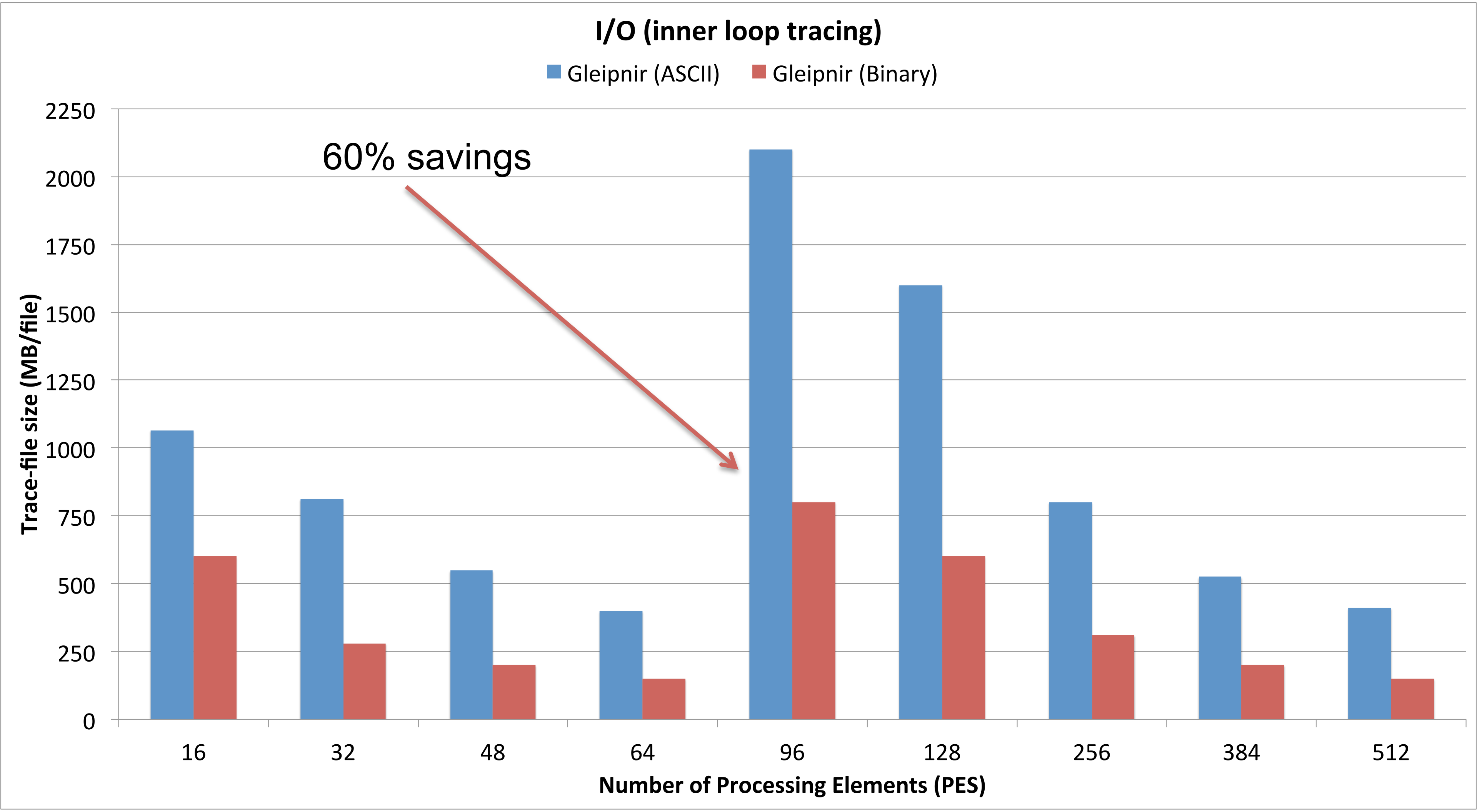
Scalability & Performance (Tracing and I/O)



Scalability & Performance (Tracing and I/O)



Scalability & Performance (Tracing and I/O)



Scalability & Performance (ASCII vs. Binary)

- The ASCII trace-line can consume up to 312 bytes although in our experience it rarely consumes over 128 bytes.

- A single iteration of 23 instructions produces ~70bytes per instruction, which means 1k loop ~70kbytes

- A single computational pass is ~4k iterations.
  - Generated ≈6.7MBs per file, and ≈100MB for a single node for 16 PEs.
  - inner loop quickly adds orders of magnitude trace data.
  - ≈1.65GB trace-files, and ≈25GB for a single node with 16PEs running.

Scalability & Performance (ASCII vs. Binary)

- The binary format is significantly smaller and consumes just 20 bytes.
  - The key difference is that the binary format stores function and variable names and tags them with an id for later look-up.

```
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;
```

Lessons learned / Experiences

- Tools for Tools (Instrumenting Valgrind-Gleipnir is difficult)
  - Required custom tools
- Cray-mpich, and huge tables do not mix well with Valgrind
  - Proposed patch and should be resolved on Oak Ridge systems.
- Valgrind is architecture sensitive, codes must avoid AMD specific optimizations (FMA4 instructions)

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 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.

- Using Gleipnir for memory tracing parallel applications is a promising technology, which has been shown to work in our Cray setting.

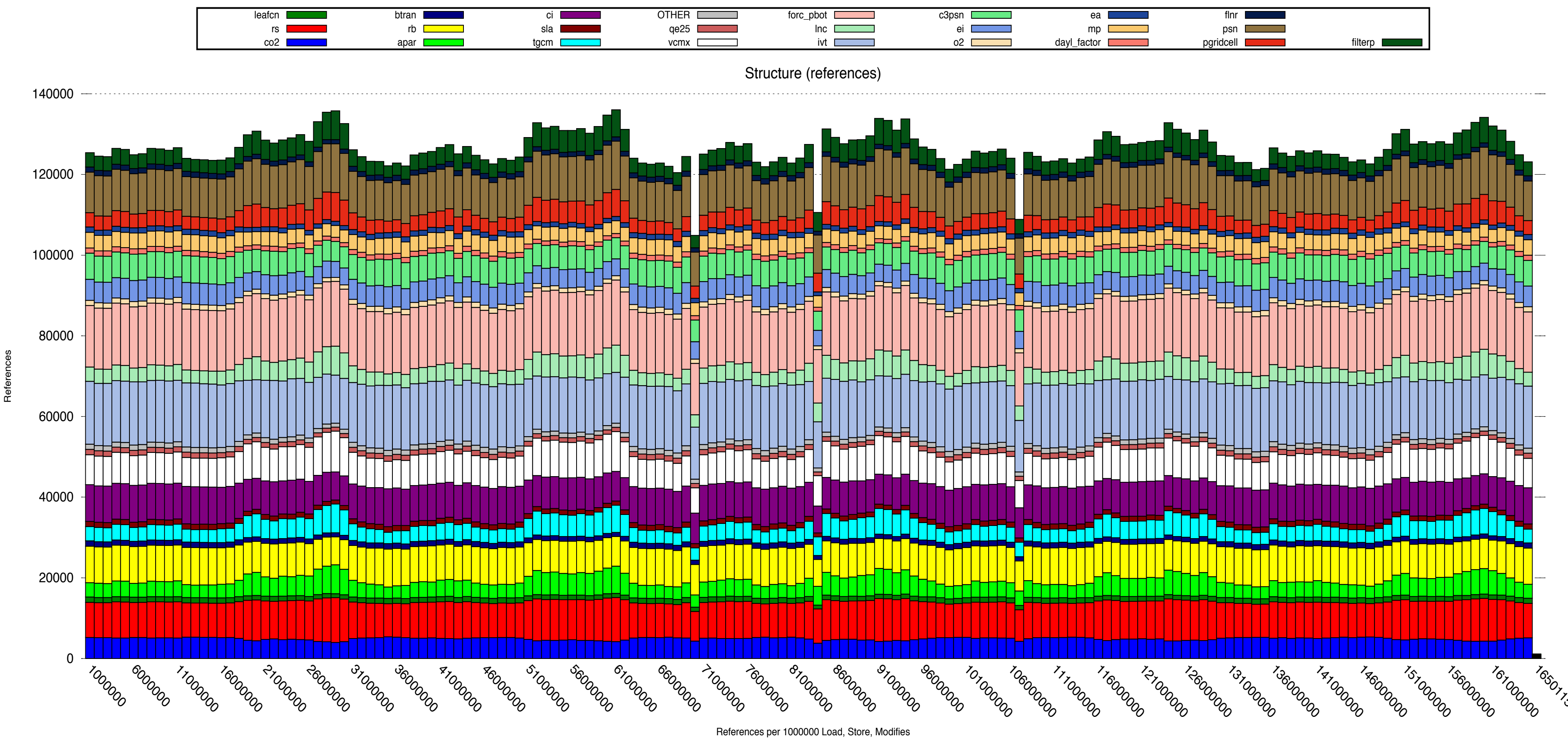
Acknowledgements

- 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.

Questions?

Blank slide

Access pattern example



Related tools and frameworks

- Valgrind's framework comes with a set of widely used tools.
  - The tool that Valgrind is most known for is Memcheck, but it also provides other tools geared towards profiling such as Cachegrind, Callgrind, and Massif.

- A similar and performance efficient tool is Pin, a dynamic binary instrumentation framework which follows the model of the popular ATOM tool.

- Within the same category we find DynInst designed for code patching and performance measurement.