

---

# **Sigil2 Documentation**

***Release 0.1.0***

**Michael Lui**

**Mar 07, 2018**



---

## Contents

---

<b>1</b>	<b>Quickstart</b>	<b>3</b>
1.1	Building Sigil2 . . . . .	3
1.2	Running Sigil2 . . . . .	3
1.3	Dependencies . . . . .	4
<b>2</b>	<b>Overview</b>	<b>5</b>
2.1	Workloads . . . . .	5
2.2	Event Generation . . . . .	6
<b>3</b>	<b>User Documentation</b>	<b>9</b>
3.1	The Analysis Backend . . . . .	9
3.2	The Profiling Frontend . . . . .	9
3.3	FAQ . . . . .	10
<b>4</b>	<b>Backend Documentation</b>	<b>11</b>
4.1	SimpleCount . . . . .	11
4.2	SynchroTraceGen . . . . .	11
<b>5</b>	<b>Frontend Documentation</b>	<b>13</b>
5.1	Valgrind . . . . .	13
5.2	DynamoRIO . . . . .	15
5.3	Intel Process Trace . . . . .	16
<b>6</b>	<b>Developer Documentation</b>	<b>17</b>
<b>7</b>	<b>About</b>	<b>19</b>
7.1	Why call it Sigil2? . . . . .	19
<b>8</b>	<b>Features</b>	<b>21</b>
<b>9</b>	<b>Installation</b>	<b>23</b>
<b>10</b>	<b>Contribute</b>	<b>25</b>
<b>11</b>	<b>Support</b>	<b>27</b>
<b>12</b>	<b>License</b>	<b>29</b>



Sigil2 is a framework for observing and analyzing applications.



This document will go through building and running Sigil2.

## 1.1 Building Sigil2

**Note:** The default compiler for **CentOS 7** and older (`gcc <5`) does not support **C++14**. Install and enable the official [Devtoolset](#) before compiling.

Clone and build Sigil2 from source:

```
$ git clone https://github.com/VANDAL/sigil2
$ cd sigil2
$ mkdir build && cd build
$ cmake{3} .. # CentOS 7 requires cmake3 package
$ make -j
```

This creates a `build/bin` folder containing the **sigil2** executable. It can be run in place, or the entire `bin` folder can be moved, although it's not advised to move it to a system location.

## 1.2 Running Sigil2

Sigil2 requires at least two arguments: the backend analysis tool, and the executable application to measure:

```
$ bin/sigil2 --backend=stgen --executable=./mybinary
```

The backend is the analysis tool that will process all the events in `mybinary`. In this example, `stgen` is the backend that processes events into a special event trace that is used in [SynchroTrace](#).

More information on backends are in [The Analysis Backend](#).

A third option `frontend` will change the underlying method for observing the application. By default, this is [Valgrind](#):

```
$ bin/sigil2 --frontend=valgrind --backend=stgen --executable=./mybinary
```

Available frontends are discussed in *The Profiling Frontend*.

## 1.3 Dependencies

PACKAGE	VERSION
gcc/g++	5+
cmake	3.1.3+
make	3.8+
automake	1.13+
autoconf	2.69+
zlib	1.27+
git	1.8+



Sigil2 is a framework designed to help analyze the dynamic behavior of applications. We call this dynamic behavior, with its given inputs and state, the **workload**. Having this workload is very useful for debugging, performance profiling, and simulation on hardware. Sigil2 was born from the need to generate application traces for [trace-driven simulation](#), so low-level, detailed traces are the primary use-case.

## 2.1 Workloads

One of the main goals behind Sigil2 is **providing a straightforward interface** to represent and analyze workloads. A workload can be represented in many ways, and each way has different requirements.

...you might represent a workload as a simple assembly instruction trace:

```
push    %rbp
push    %rbx
mov     %rsi,%rbp
mov     %edi,%ebx
sub     $0x8,%rsp
callq   4377b0 <_Z17myfuncv>
callq   4261e0 <_ZN5myotherfunc>
mov     %rbp,%rdx
mov     %ebx,%esi
mov     %rax,%rdi
callq   422460 <_ZN5GO>
add     $0x8,%rsp
xor     %eax,%eax
pop     %rbx
pop     %rbp
retq
```

...or you might represent a workload as a call graph:

...or you might represent a workload as a memory trace:

```
ADDR      BYTES
0xdeadbeef 8
0x12345678 4
0x00000000 1
...
```

...or more complex representations. Each of these representations are made up of the same event categories, albeit at different levels of granularity.

### 2.1.1 Event Primitives

Because of the variety of use-cases for analyzing workloads, Sigil2 decided to present workloads as a set of extensible primitives.

Event Primitive	Description
Compute	some transformation of data
Memory	some movement of data
Control Flow	divergence in an event stream
Synchronization	ordering between separate event streams
Context	grouping of events

The format of these events is not defined, but you can imagine that events would look like:

```
...
compute      FLOP,  add,    SIMD4
memory       write,  4B,    <addr1>
memory       read,   16B,   <addr2>
context      func,   enter,  hello_world_thread
sync         create, <TID1>
...
```

---

**Todo:** More detail is discussed futher in ???

---

## 2.2 Event Generation

Many tools exist to capture workloads:

- static instrumentation tools
  - [PEBIL](#)
  - [LLVM](#) (e.g. [Contech](#))
- DYNAMIC BINARY INSTRUMENTATION (DBI) tools
  - [Valgrind](#)
  - [DynamoRIO](#)
  - [PIN](#)
  - GPGPU specific
- HARDWARE PERFORMANCE COUNTER (HPC) sampling

- architecture-specific
- simulation probes
  - `gem5`
  - `SniperSim`
  - `Multi2Sim`
- and others

Each tool has its merits depending on the desired granularity and source of the event trace. Execution-driven simulators are great for fine-grained, low-level traces, but may be impractical for a large workload. Most DBI tools do a good job of observing the instruction stream of general purpose CPU workloads, but may not be useful when looking at workloads that use peripheral devices like GPUs or third-party IP.

Sigil2 recognizes this and creates an abstraction to the underlying tool that observes the workload. Events are *translated* into Sigil2 *event primitives* that are then presented to the user for further processing. The tool used for event generation is a Sigil2 **frontend**, and the user-defined processing on those events is a Sigil2 **backend**. Currently, backends are written as C++ static plugins to Sigil2, although there is room for expansion, given enough interest.



### 3.1 The Analysis Backend

---

**Note:** This documentation is still a WIP

---

#### 3.1.1 Getting Started with Profiling

This example will demonstrate how to get started analyzing a workload. Typically it's easier to analyze a trace file than to directly analyze a workload. That is, it's easier to generate a trace and post-process it multiple times, instead of analyzing the application on-the-fly. Parsing a trace file containing relevant data is going to be faster and more straightforward than running a workload *multiple* times and having Sigil2 filter all the potential metadata *repeatedly*.

Let's do a simple example that counts each of the event primitives:

---

**Todo:** simplecount example

- creating the backend (design to be multithreaded)
  - registering as a static plugin
  - running
- 

### 3.2 The Profiling Frontend

A *frontend* is the component that is generating the event stream. By default, this is Valgrind (mostly due to historical reasons).

While it's tempting to assume that the event generation *just works*<sup>TM</sup> you should be aware of the intrinsic nature of the chosen frontend before making any large assumptions.

### 3.2.1 Valgrind

Valgrind is the default frontend. No additional options are required. The following two command lines are equivalent.

```
$ bin/sigil2 --backend=simplecount --executable=ls -lah
$ bin/sigil2 --frontend=valgrind --backend=simplecount --executable=ls -lah
```

Valgrind is a *copy & annotate* dynamic binary instrumentation tool. This means that the dynamic instruction stream is grouped into blocks, disassembled into Valgrind's VEX IR, instrumented, and then recompiled just-in-time.

### 3.2.2 DynamoRIO

DynamoRIO is not built with Sigil2 by default. To enable DynamoRIO as a frontend, build Sigil2 using the following cmake build command:

```
$ cmake .. -DCMAKE_BUILD_TYPE=release -DENABLE_DRSIGIL:bool=true
```

DynamoRIO can now be invoked as a frontend:

```
$ bin/sigil2 --frontend=dynamorio --backend=simplecount --executable=ls -lah
```

DynamoRIO's IR exists closer to the ISA than the IR used by Valgrind. Sigil2 converts DynamoRIO IR to event primitives by inspection of each opcode.

---

**Todo:** mmm475 to fill in more details

---

### 3.2.3 Future

Additional frontends being explored include:

- LLVM-tracer
- Contech
- GPU Ocelot

## 3.3 FAQ

### 4.1 SimpleCount

#### 4.1.1 Synopsis

```
$ bin/sigil2 --frontend=FRONTEND --backend=simplecount --executable=mybinary -  
↪myoptions
```

#### 4.1.2 Description

SimpleCount is a demonstrative backend that counts each event type received from a given frontend. These events are aggregated across all threads.

#### 4.1.3 Options

No available options

---

### 4.2 SynchroTraceGen

#### 4.2.1 Synopsis

```
$ bin/sigil2 --frontend=FRONTEND --backend=stgen OPTIONS --executable=mybinary -  
↪myoptions
```

## 4.2.2 Description

SynchroTraceGen is a frontend for generating trace files for the SynchroTrace simulation framework.

Each thread detected by SynchroTraceGen is given its own output trace file, named `sigil.events-#.out`. By default, the output is directly compressed since the trace files can grow very large.

## 4.2.3 Options

`-c NUMBER`

Default: 100

Will compress all SynchroTraceGen *compute events*.

Each *compute event* will have a maximum of *NUMBER* local reads or writes

`-o PATH`

Default: `'.'`

All SynchroTraceGen output will be put in *PATH*

`-l {text,capnp,null}`

Default: `'text'`

Choose which logging framework to use.

Regardless of which logger is chosen, a `sigil.pthread.out` and `sigil.stats.out` file will be output.

`'text'` will output an ASCII formatted trace in gzipped files.

`'capnp'` will output a packed [CapnProto](#) serialized trace in gzipped files.

`'null'` will not output anything.



---

## Frontend Documentation

---

Each frontend generates one or more event streams to a Sigil2 backend analysis tool. Each frontend has its own internal representation (IR) of events, so the process of converting frontend IR to Sigil2 event primitives is different for each frontend. For example, Valgrind will disassemble each machine instruction into multiple VEX IR statements and expressions; DynamoRIO annotates each instruction in a basic block with specific attributes; the current Perf frontend only supports x86\_64 decoding via the Intel XED library.

### 5.1 Valgrind

#### 5.1.1 Synopsis

```
$ bin/sigil2 --frontend=valgrind OPTIONS --backend=BACKEND --executable=mybinary -  
→myoptions
```

#### 5.1.2 Description

Uses a heavily modified Callgrind tool, *Sigrind*, to observe Sigil2 *event primitives* and pass them to the backend. Valgrind serializes all threads in the target executable, so only one thread's event stream is passed to the backend at a time. A context switch is signaled with a Sigil2 context event. Because threads are serialized by Valgrind, the target executable is mostly deterministic.

#### 5.1.3 Options

`--at-func=FUNCTION_NAME`  
Default: (NULL)

`--start-func=FUNCTION_NAME`  
Default: (NULL)

Start collecting events at *FUNCTION\_NAME*  
If (NULL), then start from beginning of execution

`-stop-func=FUNCTION_NAME`  
Default: (NULL)  
Stop collecting events at *FUNCTION\_NAME*  
If (NULL), then stop at the end of execution

`-gen-mem={yes,no}`  
Default: yes  
Generate memory events to Sigil2

`-gen-comp={yes,no}`  
Default: yes  
Generate compute events to Sigil2

`-gen-cf={yes,no}`  
Default: no  
Currently unsupported

`-gen-sync={yes,no}`  
Default: yes  
Generate synchronization (thread) events to Sigil2

`-gen-instr={yes,no}`  
Default: yes  
Generate ISA instructions to Sigil2  
Only instruction addresses are currently supported

`-gen-bb={yes,no}`  
Default: no  
Currently unsupported

`-gen-fn={yes,no}`  
Default: no  
Sends function enter/exit events along with the function name  
Be sure to compile with less optimizations and debug flags for best results

## 5.1.4 Multithreaded Application Support

The Valgrind frontend automatically supports *synchronization events* in applications that use the **POSIX threads** library and/or the **OpenMP** library by intercepting relevant API calls.

## Pthreads

Pthreads should be supported for most versions of GCC/libc, because the Pthread API is quite stable.

Pthreads support exists for any application dynamically linked to the Pthreads library.

See *Static Library Support* for applicatons that are statically linked.

## OpenMP

Only **GCC 4.9.2** is officially supported for synchronization event capture, because the implementation of the library is more likely to change between GCC versions.

Dynamically linked OpenMP applications are not supported. Only *Static Library Support* exists.

## Static Library Support

Applications that use a static Pthreads or OpenMP library must be manually linked with the sigil2-valgrind wrapper archive. This can be found in `BUILD_DIR/bin/libsglwrapper.a`.

For example:

```
$CC $CFLAGS main.c -Wl,--whole-archive $BUILD_DIR/bin/libsglwrapper.a -Wl,--no-whole-  
↪archive
```

---

## 5.2 DynamoRIO

### 5.2.1 Synopsis

```
$ bin/sigil2 --num-threads=N --frontend=dynamorio OPTIONS --backend=BACKEND --  
↪executable=mybinary -myoptions
```

### 5.2.2 Description

---

**Note:** `-DDYNAMORIO_ENABLE=ON` must be passed to **cmake** during configuration to build with DynamoRIO support.

---

DynamoRIO is a cross-platform dynamic binary instrumentation tool. DynamoRIO runs multithreaded applications natively. This makes results less reproducible than Valgrind, however analysis is potentially faster on a multi-core architecture. This enables multiple event streams to be processed at once, by setting `--num-threads > 1`.

### 5.2.3 Options

---

**Todo:** options

---

```
--num-threads=N
```

---

## 5.3 Intel Process Trace

### 5.3.1 Synopsis

```
$ bin/sigil2 --frontend=perf --backend=BACKEND --executable=perf.data
```

---

### 5.3.2 Description

---

**Note:** `-DPERF_ENABLE=ON` must be passed to **cmake** during configuration to build with Perf PT support.

---

Intel Process Trace is a new CPU feature available on Intel processors that are Broadwell or more recent. The trace is captured via branch results. The entire trace is then reconstructed by perf by replaying the binary, including all shared library loading and context switches. A side effect of only capturing branch results is that all runtime information within the trace is lost, such as some memory access addresses; e.g. the Perf ‘replay’ mechanism does not support replaying malloc results.

For more usage details, see: [perf design document for Intel PT](#)

For more technical details see: [Intel Software Developer’s Manual Volume Three](#)

### 5.3.3 Options

---

**Note:** The `perf.data` file is generated with: `perf record -e intel_pt//u ./myexec`

If you receive ‘*AUX data lost N times out of M!*’, try increasing the size of the AUX buffer. Otherwise a significant of the portion of the trace may not be reproduced: `perf record -m,AUXTRACE_PAGES -e intel_pt//u ./myexec`

---

---

**Todo:** options

---

---

### Developer Documentation

---

---

**Todo:** section for developers  
event primitives in depth  
section for backend writing/multithreaded  
frontend event generation  
frontend IPC  
frontend synchronization capture

---

Fig. 6.1: Sigil2 Flow from Frontend to Backend  
Legend



Sigil2 comes from Drexel University’s [VLSI & Architecture Lab](#), headed by **Dr. Baris Taskin** and in collaboration with Tufts University’s **Dr. Mark Hempstead**.

The goal of Sigil2 is modular application analysis. It was formed from the need to support multiple projects that study application traces, aimed at data-driven architecture design. This has included early hardware accelerator co-design [\[SIGIL\]](#), as well as uncore design space exploration with multi-threaded workloads [\[SYNCHROTRACE\]](#) [\[UNCORERPD\]](#). Sigil2 is not interested in instrumenting the behavior of an application, but instead aims to classify events in the application and present those events for further analysis. In this way, Sigil2 does not require that each researcher have an in depth understanding of the binary instrumentation tools.

## 7.1 Why call it Sigil2?

The initial incarnation of [Sigil](#) was developed by **Dr. Siddharth Nilakantan** for his research into software-hardware co-design [\[SIGIL\]](#). He named it after Sigil, [a city in Planescape: Torment](#). He also pronounced it “sih-gul”. The current maintainer and developer of Sigil2, [Michael Lui](#), has kept the name and pronunciation for historical purposes. However, all of the underlying code and infrastructure has been rewritten and enhanced.

---





## CHAPTER 8

---

### Features

---

- Flexible application analysis
  - Use multiple frontends for capturing software workloads like Valgrind and DynamoRIO
  - Use custom C++14 libraries for analyzing event streams
- Platform-independent events
  - Straight-forward and extensible format, simplifying analysis



## CHAPTER 9

---

### Installation

---

See the *Quickstart* for information installation instructions.



## CHAPTER 10

---

Contribute

---

Source Code: <https://git.io/sigil2>

Issue Tracker: <https://github.com/VANDAL/sigil2/issues>



## CHAPTER 11

---

### Support

---

Please contact our mailing list for any issues or concerns: [sigil2@googlegroups.com](mailto:sigil2@googlegroups.com)





## CHAPTER 12

---

### License

---

This project is licensed under the [BSD3 license](#).



---

## Bibliography

---

- [SIGIL] S. Nilakantan and M. Hempstead, “*Platform-independent analysis of function-level communication in workloads*”, 2013 IEEE International Symposium on Workload Characterization (IISWC), pp. 196 - 206, 2013.
- [SYNCHROTRACE] S. Nilakantan, K. Sangaiah, A. More, G. Salvatory, B. Taskin and M. Hempstead, “*Synchro-trace: synchronization-aware architecture-agnostic traces for light-weight multicore simulation*”, 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 278 - 287, 2015.
- [UNCORERPD] K. Sangaiah, M. Hempstead and B. Taskin, “*Uncore RPD: Rapid design space exploration of the uncore via regression modeling*”, 2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 365 - 372, 2015.