Simulation Guided Memory Analyzer

SiGMA

Simone Sbaraglia
Advanced Computing Technology Center
IBM Research
sbaragli@us.ibm.com
Phone: +1-914-945-2546
Fax: +1-914-945-4269

Version 2.5 - July 19, 2004

Table of Contents

---

1. The Simulation Guided Memory Analyzer

The Simulation Guided Memory Analyzer (SiGMA) is a toolkit designed to help programmers to understand the precise memory references in scientific programs that are causing poor utilization of the memory subsystem. Fine-grained information such as this is useful for tuning loop kernels, understanding the cache behavior of new algorithms, and to investigate how different parts of a program compete for and interact within the memory subsystem.

The SiGMA toolkit consists of a pre-execution tool that locates and instruments all instructions that refer to memory locations, a runtime data collection tool that performs a highly efficient lossless compression of the stream of memory addresses generated by the instrumentation, and a set of simulation and analysis tools that process the compressed memory reference trace to provide programmers with tuning information.

---

2. Requirements

SiGMA was developed for performance analysis of Fortran and C applications, compiled with IBM XLC and/or XLF compilers, running on IBM Power3 and Power4 systems, under AIX 5L.

In order to use SiGMA, users are required to define an environment variable $SIGMA, which points to the path of the SiGMA installation.

---

3. Instrumentation

Instrumentation is performed automatically at the binary level to capture runtime information about memory operations. It uses a binary instrumentation approach, as opposed to source instrumentation, so that the gathered data reflects actual memory references generated by optimizing compilers. In addition, the source code is not required as long as the debugger information is present in the binary (*i.e.* the binary must have been produced with the “*-g*” option).

The instrumentation task is accomplished with the *"sigmaInst"* application, that can be found in the *$SIGMA/bin* directory.

To instrument a binary *appbin*, the "*-d*" option of sigmaInst must be used, as in

```
sigmaInst –d appbin
```

This will produce an instrumented binary *appbin.inst*. By default, the application is instrumented to run a direct simulation (see Section 4 for a description of the available execution modes). To generate a trace file instead, the command would be:

```
sigmaInst –d –dback tr appbin
```

Other useful sigmaInst options are:

-dlink:	pass options to the linker: if the application was linked with any special argument, this argument has to be passed again to sigmaInst. For example, if the application was linked with the arguments "*-lessl –bmaxdata:0x60000000*", then instead of *"sigmaInst –d appbin"*, the command would be: `sigmaInst –d –dlink "-lessl –bmaxdata:0x60000000" appbin`
-dmpi:	use this flag to instrument an MPI application. Each MPI task will generate a separate md profile (or trace file), identified by its MPI task id.
-domp:	instrument an OpenMP application. Multiple streams will be generated, with no synchronization.
-p descr-file lib app:	insert user-probes in the binary. Please refer to the file "up.pdf" in the SiGMA distribution for the details on user probes
-dyn:	use this flag to enable dynamic tracing (see later for explanation)
-sdyn:	use this flag to enable slow dynamic tracing (see later for explanation)
-dstatic:	relink the binary statically, to instrumented functions and load/store operations in shared libraries.
-dbuf n:	specify the buffer size in pages (default is 1). A larger buffer leads to faster execution, especially with large applications, but will require more memory to run.

-dfunc f1,…,fn:	specify which functions to instrument. By default, all the functions that are statically linked into the executable *appbin* will be instrumented. The function names must be typed exactly as they appear in the binary. Use *"sigmaInst –funct appbin"* to display all the functions in the binary.
-dfile f1,…,fn:	specify which files to instrument. If a file is selected, all the functions in that file will be instrumented. Use *"sigmaInst –filet appbin"* to display all the files in the binary.
-dfuncf <filename>:	read the functions to instrument from the file *filename*.
-dfilef <filename>:	read the files to instrument from the file *filename*.
-dxfunc f1,…,fn:	specify which functions to exclude from the instrumentation. Can be combined with the other selection options.
-dxfile f1,…,fn:	specify which files to exclude from the instrumentation. If a file is excluded, all the functions in that file are excluded from the instrumentation, unless a *–dfunc* statement that selects them appears later in the command line.
-dxfuncf <filename>:	read the functions to exclude from the instrumentation from the file *filename.*
-dxfilef <filename>:	read the files to exclude from the instrumentation from the file *filename*.
-dout outbin:	specify the instrumented binary name (*appbin.inst* by default).

Please run *sigmaInst* with no arguments for the full list of options.

For examples of how to instrument an application refer to the *examples* directory in *$(SIGMA)/doc/examples*.

Dynamic Tracing under program control

In addition to statically specifying which functions or files to instrument, using the *–dfunc* or *–dfile* flags of sigmaInst, it is also possible to annotate the application to start and stop the sigma tracing dynamically under program control.

This functionality can be very useful to selectively analyze one portion or iteration of the code, or to skip the initialization steps and more in general in all the situations where the code to analyze is not enclosed in a set of functions or changes dynamically while the application runs.

In order to use this facility, #include the header file $SIGMA/include/signal_sigma.h in your application and link the application with $SIGMA/lib/libsigma.a.

The binary application has then to be instrumented using the *–dyn* or *–sdyn* option of *sigmaInst*. The *–dyn* mode is faster than the *–sdyn* mode since it is based on overwriting the program image in memory, and is therefore preferable. The *–sdyn* mode is a simpler implementation of the dynamic tracing facility that incurs some overhead in running the application (both with tracing on and off). This mode is to be used only for testing and debugging.

The application can then call *signal_sigma(TRACE_ON, section_id)* to activate the SIGMA tracing and *signal_sigma(TRACE_OFF, section_id)* to deactivate it.

section_id is an integer used to identify the current code section. By using different identifiers it is possible to profile different portions of the code (for example different iterations in a loop) separately. A separate set of *.md* and *.viz* files is produced for each section id.

Multiple sections are currently supported only with the fast *–dyn* dynamic mode. The *–sdyn* dynamic mode just ignores the section id and assumes section id = 0.

Similarly, when dynamic tracing mode is not active the results are dumped under section id 0.

Please refer to $SIGMA/doc/examples/signal_sigma_example for an example of use of the dynamic tracing facility.

The signal_sigma interface is a flexible mechanism to communicate signals or commands to the SIGMA infrastructure, and can be used to generate commands like "dump the state of the cache now", "reset the cache" as well as changing memory parameters dynamically (turn prefetcher on and off under program control etc).

The commands that are currently supported are:

- TRACE_ON: activate the SIGMA tracing
- TRACE_OFF: deactivate the SIGMA tracing
- ERASE_MEM: erase contents of memory
- ERASE_PREF: erase all prefetcher streams (reset prefetcher)
- ERASE_COLD: erase cold misses history (reset cold misses)
- RESET_MEM: completely reset memory (= ERASE_MEM + ERASE_PREF + ERASE_COLD)

Please note that if the SIGMA tracing is turned off, when it is turned on again we assume that the memory contains whatever was there at the moment that the tracing was turned off (the state is “frozen” when the tracing is turned off, no cleanup occurs).

In order to reset the cache to a clean (cold) state, use the commands above.

The *signal_sigma* function prototype is

void signal_sigma(int code, int arg)

where *code* is one of the above (TRACE_ON, TRACE_OFF etc) and *arg* is an integer argument currently used only if code is TRACE_ON or TRACE_OFF to identify the code section. The *arg* value is disregarded in all other cases.

---

4. Running the Application

SiGMA has two modes of operation: *"Direct"* or *"FromTrace"*. In Direct Mode execution the simulation is performed "on-the-fly" when running the instrumented application, while in the FromTrace mode, the user runs the instrumented application to generate a trace file and then runs the simulator, which takes the trace file as input. The former is faster since it does not require generating and reading the trace file, but might be inconvenient if the user needs to test with different architecture specifications or data structure configurations; since the application has to be re-executed for each new simulation. In this case, FromTrace would be more effective, since one can generate the trace file once and then experiment with different architectures or different data structure layouts.

4.1. Direct Mode

In Direct mode execution the user generates an instrumented version of the application as described in Section 3, then runs the instrumented binary (called *appbin.inst* here), supplying the architecture specification file, as in *./appbin.inst –sigma architecture-file.*

The machine specified in the architecture file is then simulated while the application runs, and the metrics are collected in the file *appbin.inst.mp_task.secN.machine.md*, where *machine* is the name of the machine as specified in the architecture file (see Section 4.3 for a description of the syntax of the architecture file), *mp_task* is the MPI task or 0 in the case of serial applications and *N* is the section id as specified in *signal_sigma(TRACE_ON, section_id)* or 0 if dynamic tracing is not used.

Examples of architecture specifications for Power3 and Power4 can be found in *$(SIGMA)/archs/sigmaArchPower3* and *$(SIGMA)/archs/sigmaArchPower4.*

If the application requires any arguments, they can be supplied before the *–sigma* switch, as in:

```
./appbin.inst application-arguments –sigma architecture-file
```

Every argument preceding the *"-sigma"* switch will be passed uninterpreted to the application *appbin.inst.*

It is also possible to apply a padding technique to the simulation by means of the *"-padding"* option, as in:

```
./appbin.inst –sigma architecture-file -padding paddingfile
```

where *paddingfile* is the specification file for padding (see Section 4.4 for the syntax of the padding specification file).

4.2. From Trace Mode

In FromTrace mode, the user instruments the application to generate a compressed trace file first (by using the *"-dback tr"* option of sigmaInst). The trace file can then be used as input to the simulator at a later time.

After instrumenting the application with *sigmaInst –dback tr*, the user runs the application (without supplying any architecture file, just the application-specific arguments, if any). As a result of this run, a compressed trace file *"appbin.mp_task.cfz"* will be generated.

It is then possible to run the memory simulation for a machine specified in the architecture file *architecture-file* by running:

```
$(SIGMA)/bin/sigmaMd appbin.mp_task.cfz –sigma architecture-file.
```

Again, it is possible to use a padding technique with:

```
$(SIGMA)/bin/sigmaMd appbin.mp_task.cfz –sigma architecture-file
-padding paddingfile.
```

Run *sigmaMd* with no arguments for a list of all options.
Please note that when generating a trace file in dynamic mode (*i.e. when using the flags "–dyn –dback tr" at the same time*) the section id is disregarded and one single trace file for all the sections is generated.

4.3 The Architecture File

The architecture file specifies the memory configuration to be used in the simulation. The user can specify parameters such as number of cache level, line size, associativity etc. An example of an architecture file is presented next.

```
##type name   keyword       number-of-machines
system s1     machines              1
##type  name   keyword #caches keyword #tlbs prefType ArithOpLatency
MemHitLatency Frequency(Mhz)
machine power3 caches  2       tlbs    1     p3       1              28
375
##type name   szBts  lnSzBts  assoBts prefch replace writePolicy
computeCold  Latencies(Load Store)
cache  l1     16     7        7       no     lru     writeback    yes
2       2
cache  l2     22     7        0       no     lru     writeback    no
6       6
tlb    Tlb    20     12       1       no     lru     none         no
112
# end
```

The lines starting with "#" are comments. The first non-comment line of the file defines the set of machines specified in the architecture file (called a *system*) and specifies the number of machines composing the system. The syntax of the first line is the following, where we indicate, with <*parameter*> a user supplied parameter and with *keyword* a keyword:

- the keyword *system*
- <*string*>*:* the name of the system
- the keyword *machines*
- <*integer*>*:* the number of machines in the system

Then, for each machine of the system, the user specifies the memory layout of the machine followed by the specification of each memory level.

In our example, the memory layout of the machine is specified by the line:

```
##type  name   keyword #caches keyword #tlbs prefType ArithOpLatency
MemHitLatency Frequency(Mhz)
machine power3 caches  2       tlbs    1     p3       1              28
375
```

and the various levels of the memory hierarchy are specified by the lines:

```
##type name   szBts  lnSzBts assoBts prefch replace writePolicy
computeCold  Latencies(Load Store)
cache  l1     16     7       7       no     lru     writeback    yes
2       2
cache  l2     22     7       0       no     lru     writeback    no
6       6
tlb    Tlb    20     12      1       no     lru     none         no
112
# end
```

The syntax for the line that specifies the memory layout is the following:

- the keyword *machine*
- <*string*>*:* name of the machine (no spaces)

- the keyword *caches*
- *<integer>:* number of cache levels
- the keyword *tlbs*
- *<integer>:* number of TLBs (only one supported with current release)
- one of the keywords *p3* or *p4* indicating which prefetcher should be used (Power3 or Power4)
- *<integer>:* latency (cycles) of an arithmetic operation (not used in current release)
- *<integer>:* latency (cycles) of bringing data from main memory (memory hit)
- *<integer>:* frequency of the machine, in Mhz.

Each cache level has to be described by a line of the form:

```
cache l1  16  7 7  no  lru  writeback yes  2  2
```

whose syntax is the following:

- the keyword *cache*
- *<string>:* the name of the cache level (not used at the moment)
- *<integer>:* cache size in bits, i.e. if this number is 16 as in our example, the cache size is 2^16=65536bytes.
- *<integer>:* line size in bits
- *<integer>:* associativity in bits
- one of the keywords *yes* or *no* indicating whether prefetching should be enabled for the cache level
- one of the keywords *lru*, *fifo* or *rand* specifying the replacement algorithm (least-recently used, first-in-first-out or random)
- one of the keywords *writeback* or *writethru*, to specify the cache type
- one of the keywords *yes* or *no* to specify whether cold misses should be computed for the current cache level
- *<integer>:* the latency (in cycles) of a load operation from the cache level
- *<integer>:* the latency (in cycles) of a store operation from the cache level

The lines describing each cache level must come in order, i.e. L1, followed by L2, L3 etc. After these lines there must be a line for each TLB level (however, there is only one level supported at this time). This line has the form:

```
tlb   Tlb   20    12    1     no    lru   none      no      112
```

and the syntax is:

- the keyword *tlb*
- *<string>:* the name of the TLB level (not used at the moment)
- *<integer>:* TLB size in bits
- *<integer>:* page size in bits, in our example a page is *2^12=4096* bytes.
- *<integer>:* associativity in bits

- one of the keywords *yes* or *no* indicating whether prefetching should be enabled for the TLB level
- one of the keywords *lru*, *fifo* or *rand* specifying the replacement algorithm (least-recently used, first-in-first-out or random)
- the cache type is ignored for TLBs
- one of the keywords *yes* or *no* to specify whether cold misses should be computed for the current TLB level
- *<integer>:* the latency (in cycles) of a load operation from the TLB level

Note that the TLB size is not to be interpreted as bytes but as number of entries, indicating that if the TLB has *N* entries, and the page size is *2^l*, then the size *n* of the TLB in bits is such that: *N=2^(n-l)*. In our example, the page size is 4096 bytes (*l* = 12*)*, the TLB has 256 entries, therefore *n* is such that: *256=2^(n-12)*, i.e. *n* = 20. For example, a TLB description to simulate large pages on the Power4 would be:

```
tlb   Tlb   34    24    2     no    lru   none      no      700
```

Multiple machines can be simulated at the same time, as shown in the following example:

```
##type name   keyword       number-of-machines
system s1     machines              2
##type  name   keyword #caches keyword #tlbs prefType ArithOpLatency
MemHitLatency Frequency(Mhz)
machine power3 caches  2       tlbs    1     p3       1              28
375
##type name   szBts  lnSzBts  assoBts prefch replace writePolicy
computeCold     Latencies(Load Store)
cache  l1     16     7        7       no      lru    writeback     yes
2       2
cache  l2     22     7        0       no      lru    writeback     no
6       6
tlb    Tlb    20     12       1       no      lru    none          no
112
# end
##type  name   keyword #caches keyword #tlbs prefType ArithOpLatency
MemHitLatency Frequency(Mhz)
machine power4 caches  3       tlbs    1     p4       1
250            1300
##type name   szBts   lnSzBts  assoBts prefch replace writePolicy
computeCold  Latencies(Load Store)
cache  l1     15      7        1       yes    fifo    writethru    yes
2       4
cache  l2     20      7        3       no     lru     writeback    no
12      14
cache  l3     27      9        3       no     lru     writeback    no
102     104
tlb    Tlb    22      12       2       no     lru     none         no
700
# end
```

4.4. Data Structure Padding

The padding specification file defines parameters for data structure padding, which can be either *"external"* or *"internal"*. External padding of an array *"A"* with *"N"* bytes means that a *dummy* data structure of size *N* bytes is inserted after *A* in the data structure layout, while internal padding of a matrix means that one of the *inner* dimensions of the matrix is increased by the specified amount.

The following is an example of a padding specification file:

```
#padding command file
INTERNAL u 1 1
INTERNAL v 1 3
EXTERNAL p 0 8
```

Lines beginning with *"#"* are comments. The padding command file defines a list of padding specifications. The various padding specifications are applied in the order in which they appear in the file. The syntax of each specification line in the padding file is the following:

- one of the keywords *INTERNAL* or *EXTERNAL*to specify internal or external padding
- *<string>:* the name of the array to pad. The array must appear in the ".addr" file for the application. This means that only external C variables and Fortran common block variables can be padded with the current release.
- *<integer>:* the dimension to pad for internal padding. This field is ignored for external padding.
- *<integer>:* the amount of padding to be applied. This field is interpreted differently in internal and external padding: in internal padding this refers to the amount by which the specified dimension has to be increased. For example the line: "`INTERNAL v 1 3`" specifies that the first dimension of the matrix *"`v`"* has to be increased by *3*. So, if *"`v`"* is, for example, a *3x3* matrix, it would become a *6x3* matrix.

  In external padding, this number is interpreted as number of *dummy* bytes to be appended to the data structure.b

5. Output

The output file *appbin.inst.mp_task.secN.machine.md* contains the SiGMA memory profile for the application, obtained by simulating the machine *"machine"* described in the architecture file. The output file consists of four sections:

1. The first section, denoted by *METRICS BY CACHE LEVEL*, summarizes the results for each level of cache and for the TLB. For each cache level, the

Architectural Information section summarizes the cache level characteristics defined in the architecture file, followed by the counters and metrics for the cache level.

2. The second section, denoted by *METRICS BY FUNCTION*, reports the metrics for each function in the source code. The functions are sorted in decreasing order based on an estimate (called *"memtime"*) of the time spent in the memory communication, defined as follows:

   memtime = [memAcc*memLat +TLBmiss*TLBLat + SUMi=1,n(LdiHits*LdiLat +StiHits*StiLat)]/Frequency

   where *"LdiHits"* and *"StiHits"* are the number of loads and stores hits at level "i", respectively, and the latencies *("LdiLat"*, *"StiLat"*, *"TLBLat",* and *"memLat")* parameters are the ones supplied by the user in the architecture file. The user can consider this section as a *"memory profile"* of the functions in the code. The first functions are the most expensive in terms of the *"memtime"* metric.

3. The third section, denoted by *METRICS BY DATA,* shows the collected metrics for each data structure, sorted based on the *memtime* of each data structure.
4. The fourth section, denoted by *METRICS BY DATA@FUNCTION*, shows the collected metrics for each function and for each data structure within the function. Again, both the functions and the data structures within the functions are sorted based on the *memtime* metric.

The output files *appbin.inst.mp_task.secN.machine.data.viz* and *appbin.inst.mp_task.secN.machine.func.viz* can be visualized using the *peekperf* GUI (*peekperf appbin.inst.mp_task.secN.machine.data.viz* or *peekperf appbin.inst.mp_task.secN.machine.func.viz*).
The *.data* file provides a data-centric view of the metrics, similar to the *METRICS BY DATA* section of the ASCII profile, whereas the *.func* file provides a control-centric view of the metrics, similar to the *METRICS BY FUNCTION* view of the ASCII profile.

---

6. Derived Metrics Description

In addition to presenting the raw counter data, SiGMA also computes the following derived metrics, which are based on the architecture parameters used as input to the simulator:

- *Accesses/Misses* = Number of Accesses / Number of Misses
- *Hit Ratio* = Number of Hits / Number Accesses * 100
- *L2 Traffic* = Number of L2 Accesses * L1 Line Size
- *L3 Traffic* = Number of L3 Accesses * L2 Line Size
- *Mem Traffic* = Number of Mem Accesses * L3 Line Size

- *Estimated Latency* (cache) = Number of Hits * Latency / Frequency
- *Estimated Latency* (TLB) = Number of Misses * Latency / Frequency
- *Estimated Latency* (mem) = Number of Accesses * Latency / Frequency

Each one of these metrics can appear separately for load and store operations. For example the *Hit Ratio* is reported as a total (Load + Store) and separately as a *Load Hit Ratio* and *Store Hit Ratio*.

---

7. Known Limitations

64-bit applications are not supported.

---

8. Release History

Version 2.5:

- Implemented the "user probes" feature, that allows users to insert their probes into the binary to instrument loads and stores, intercept and replace functions and so on. The user's probes will have access to the details of the cache simulation as well as the symbolic mapping. Please refer to the "up.pdf" manual in the SiGMA distribution for details and to the example directory for example of use.
- Implemented the *–domp* flag to instrument OpenMP applications.
- Added the *–dstatic* flag that allows to transparently relink the binary statically, in order to instrument shared libraries and/or to intercept and replace (via the user probes feature) functions in shared libraries.

Version 2.4:

- Added support for multiple code sections when using the dynamic *–dyn* mode. It is now possible to profile different code sections separately by specifying a section id when calling signal_sigma(TRACE_ON) or signal_sigma(TRACE_OFF).
- Added a prototype of the data-flow model, to analyze the flow of data between subroutines and identify the most memory intensive functions. The tool can be activated with the *–dback df* flag of *sigmaInst* or by running *sigmaDf* in fromtrace mode.
  For example:

  ➢ *sigmaInst –d –dback df appbin*
  ➢ *./appbin.inst*

Version 2.3:

- Implemented a more efficient dynamic tracing facility, based on overwriting the binary image in memory. This facility is activate with the *–dyn* switch of *sigmaInst*. The previous dynamic tracing mode present in version 2.2 is still accessible with the *–sdyn* option.

Version 2.2:

- Added dynamic tracing facility to turn tracing on and off under program control
- Added support for MPI applications. Each MPI task generates a separate profile and trace file. To instrument an MPI application it is necessary to specify the *–dmpi* flag of sigmaInst.
- Fixed memory inefficiency when dealing with dynamic memory tracing, that could lead to program crashes under certain conditions.
- Fixed several minor bugs.

Version 2.1.3:

- Major reengineering of the instrumentation approach: the instrumentation is now performed at the binary level. Source code not required anymore.
- Added support for large applications linked with the *–bmaxdata* flag.
- Added support for local variables.
- Added Fortran90 support.
- Added support for dynamically allocated C and F90 variables.

Version 1.1.2:

- Fixed a problem on "sigmaInst" to reduce the number of "Unknown" symbol types.
- Fixed bug on "sigmaInst" of dumping some global variables with address "0".
- Fixed a problem of identifying variables defined as "file-global" in C programs.
- Fixed problem with variables being skipped because of file names not being recognized.
- Extended sigmaCompile to also handle the following extensions: ".cc", ".f90", ".F", and ".C".
- Fixed sigmaCompile bug in parsing base file name (file names with "." Were being lost).
- Fixed problem of SiGMA not compiling properly with certain variable types ("M" types).
- Fixed documentation (this file) for corrected display of formulas in Sections 4.3 and 5.

Version 1.1:

- Initial Release

Top of Page