



Advanced Computing Technology Center

SIGMA

A Software Infrastructure to Guide Memory Analysis

Simone Sbaraglia
sbaragli@us.ibm.com

AGENDA

- Why SIGMA?
- SIGMA Highlights
- The SIGMA Infrastructure
- SIGMA Backends
- SIGMA User-Probes
- Next steps
- Questions/Answers

Why SIGMA?

- Efficient utilization of the **memory subsystem** is a critical factor to obtain performance
- None of the **memory analysis tools** currently available combines source-level information, low overhead and flexibility (what-if questions and architectural changes):
 - **hpm**: cannot relate to source level structures (such as data structure names, sizes, function names and line numbers) or do “what-if” analysis
 - **System tools (eprof, tprof)**: fixed account of statistics, no selection of program and data segments, no architecture changes
 - **Simulators (mambo)**: detailed system simulation but high overhead
 - **Program annotations**: require knowledge of the source code, time consuming
- Current **instrumentation tools** (DPCL, Dyninst) do not provide symbolic events and cache simulation to help users develop memory profiling tools.
 - Transparently provide the probe function with **symbolic information** and the results of the **cache simulation**
 - Intercept the program execution based on a **symbolic specification** (“invoke the probe every time the array A is touched”)
 - Intercept the program execution based on **performance events** (“invoke the probe on each L1 miss for the array A”)

SIGMA Highlights

Infrastructure for instrumentation, tracing, performance measurement and projection based on:

Automatic Instrumentation Engine:

- Operate on the binary
- Allow Compiler Optimization
- Allow injection of arbitrary user probes
- Transparently provide cache simulation and symbolic mapping
- Dynamic instrumentation capabilities
- Support instrumentation of parallel (shared-memory, MPI) applications

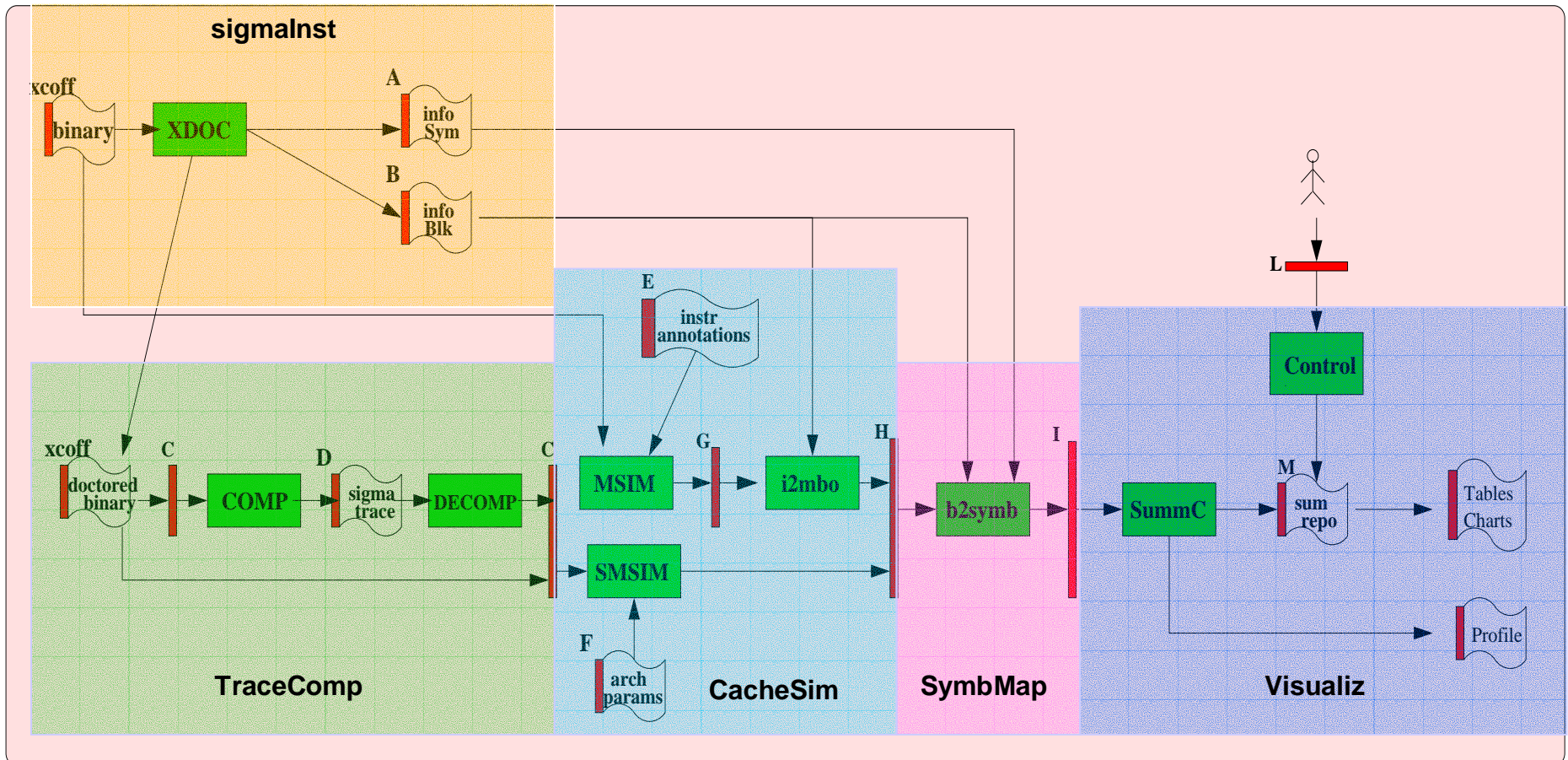
Symbolic Mapping:

- Map each instruction back to (function, lineNumber)
- Map each memory reference back to the data structure name
 - Support Global Variables, Stack Variables,
Dynamically Allocated Memory, Modules etc

Cache Simulation:

- Memory Architectural Changes
- Implement IBM Prefetching Algorithms
- Simulate Program and data-structure changes (padding)

The SIGMA Infrastructure



sigmaInst:

Binary instrumentation tool

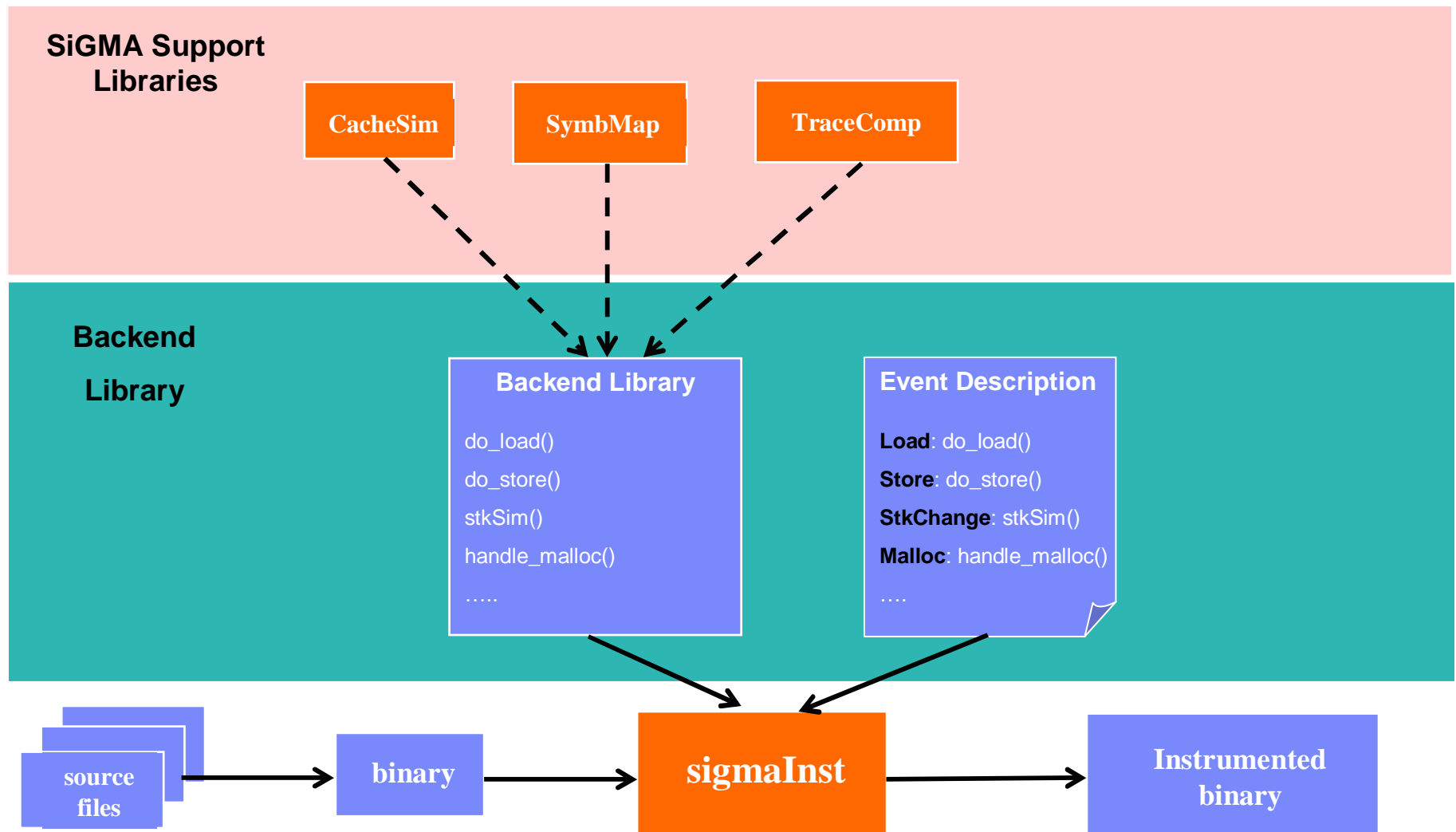
■ Features:

- Binary rewrite + dynamic patching approach
- Insert user-supplied probes at the instruction level
- Provide symbolic information
- Transparently provide access to cache simulation results
- Activate/Deactivate instrumentation dynamically and under program control
- Support MPI applications
- “Almost” supports shared-memory applications

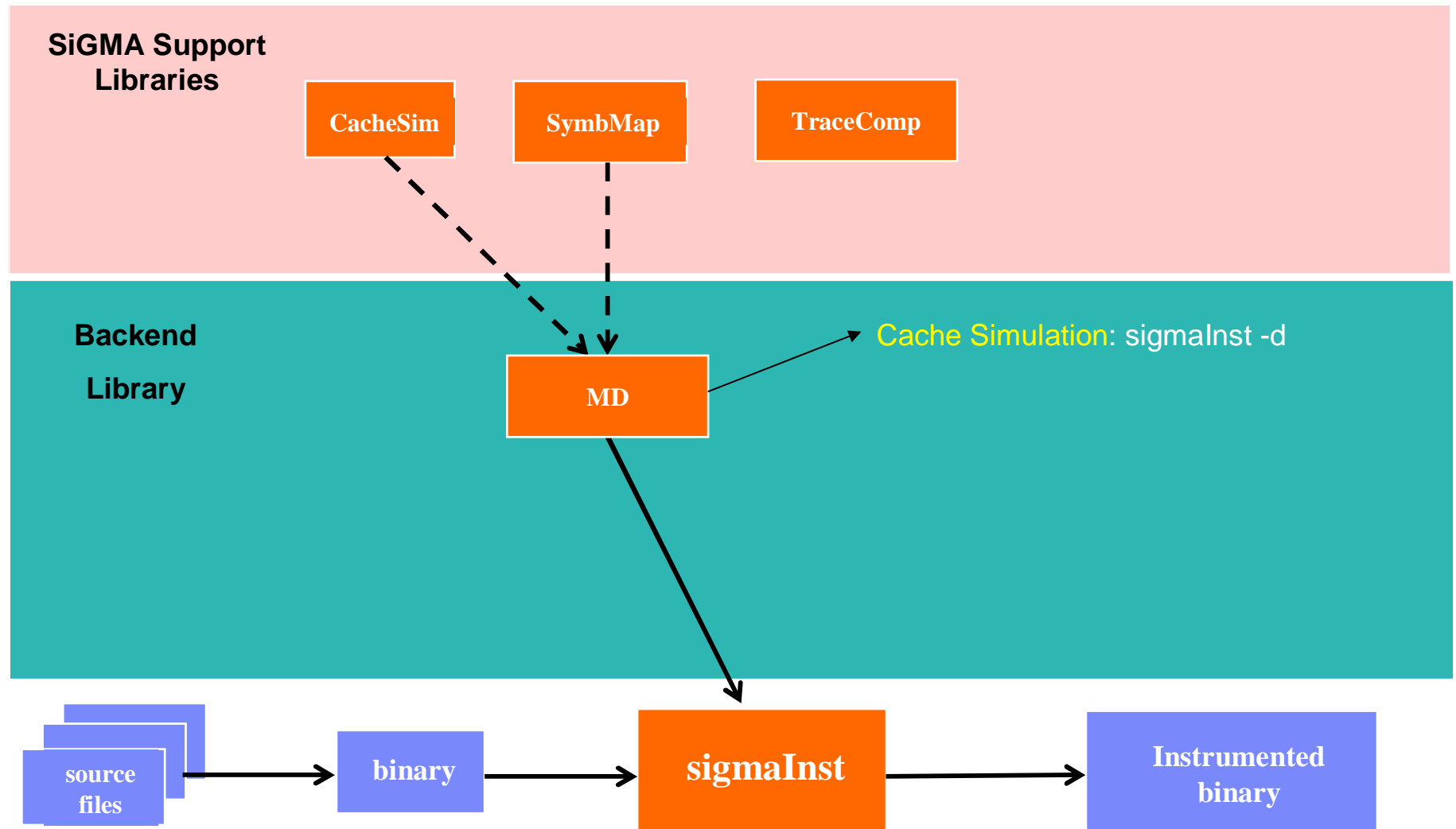
■ Limitations:

- C and Fortran programs
- 32-bit applications
- Cache simulation for single-threaded apps (work in progress...)

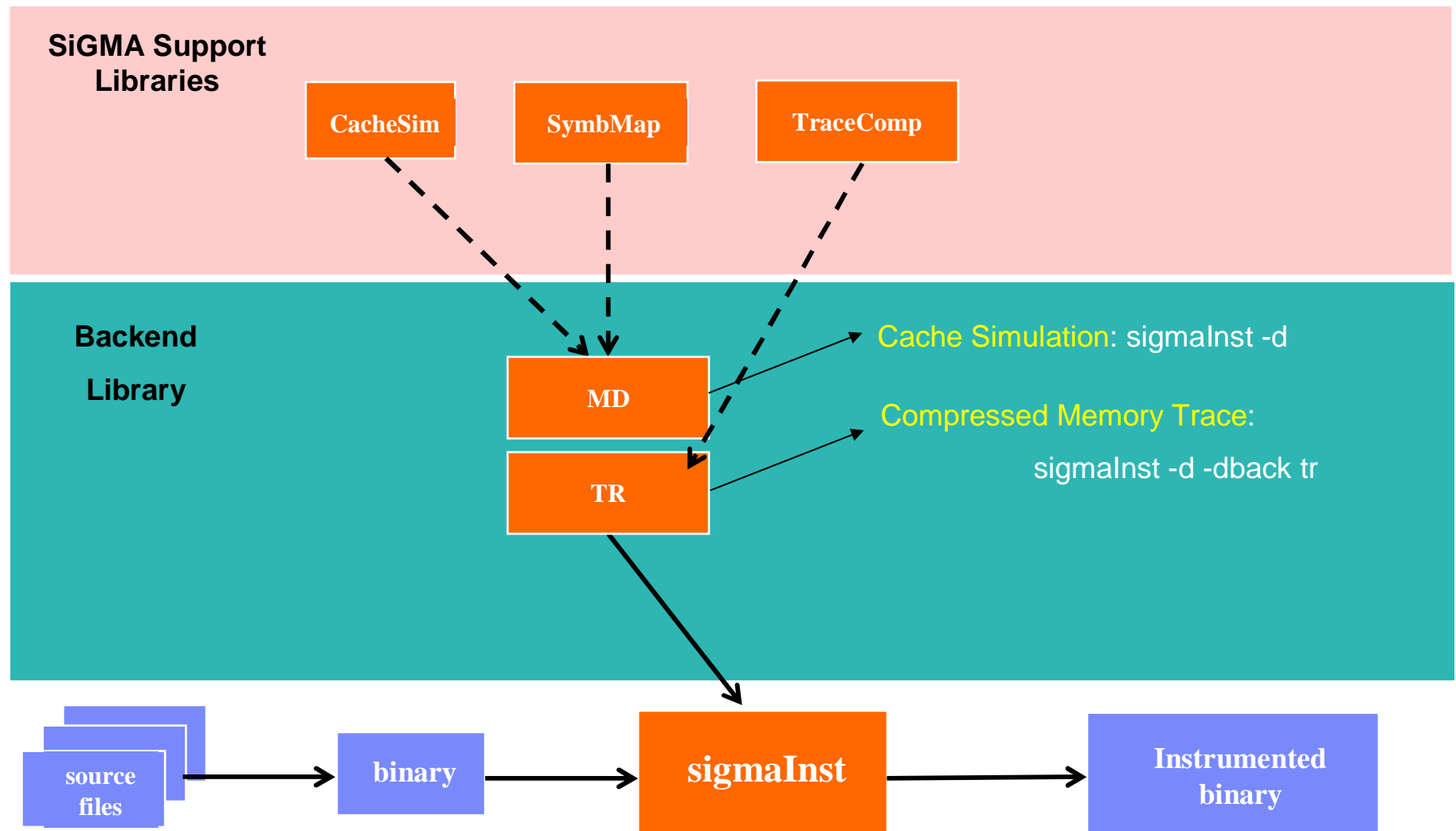
SIGMA Approach



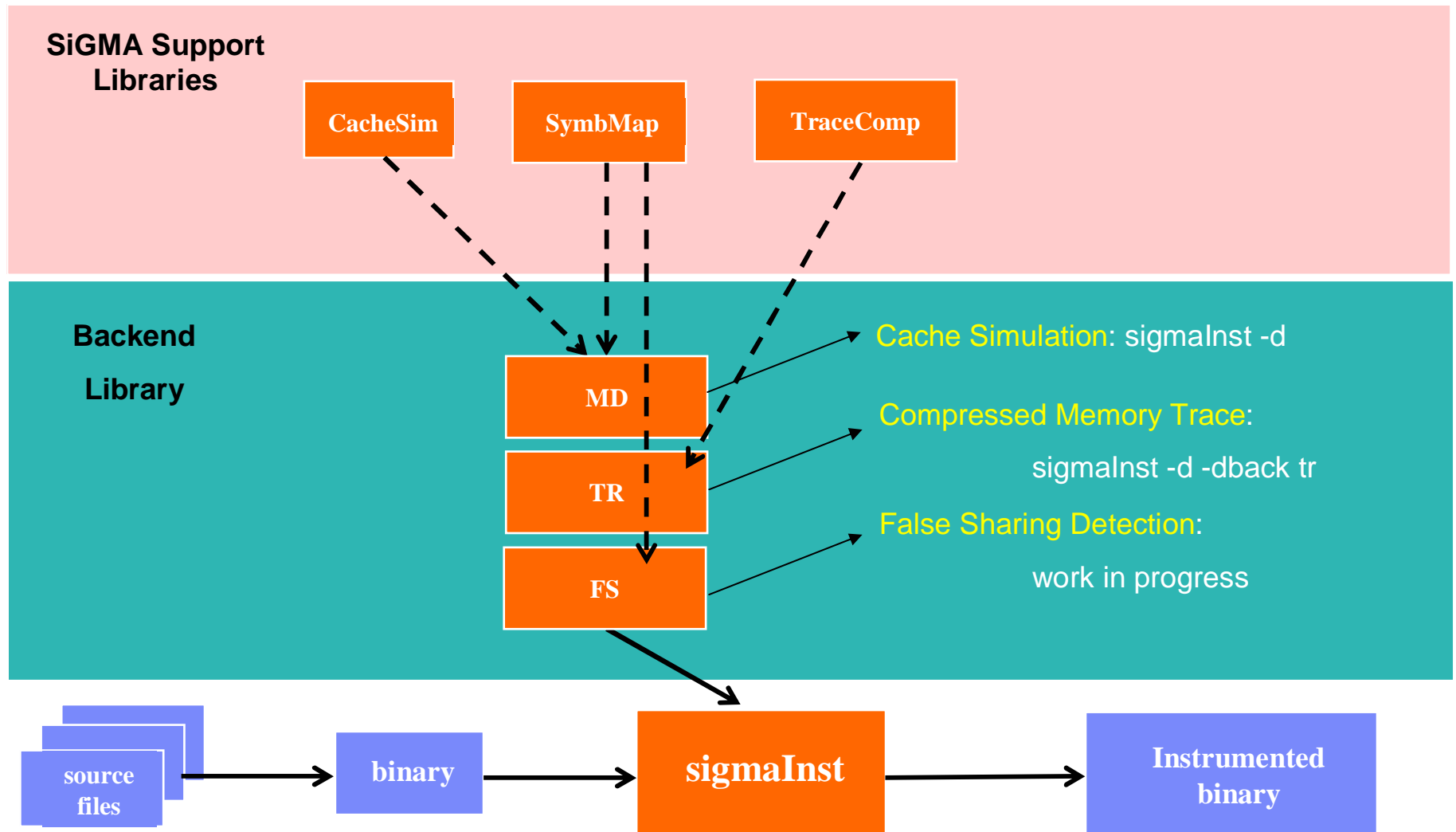
The SIGMA Backends: MD



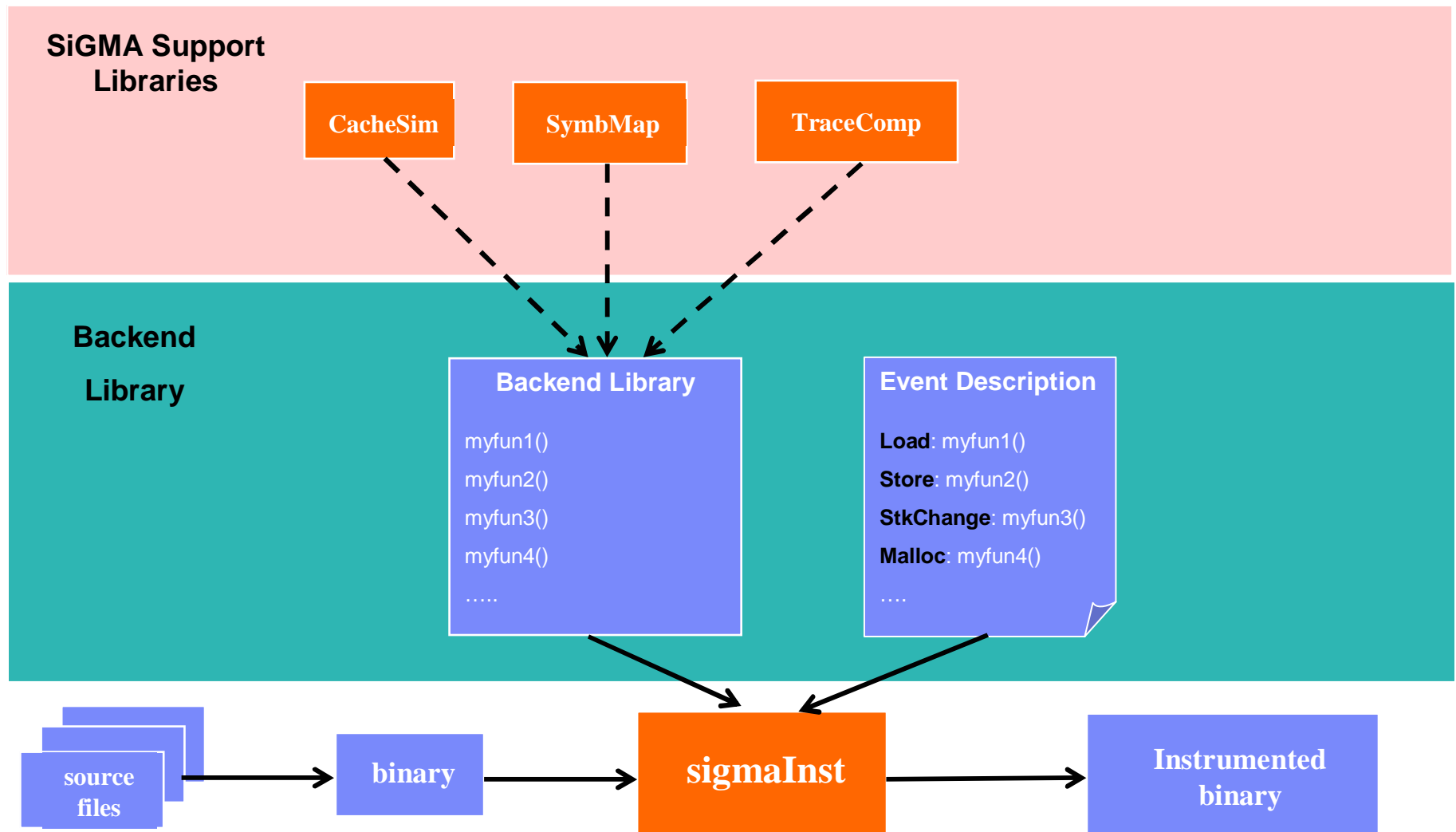
The SIGMA Backends: TR



The SIGMA Backends: FS



User-Defined Backend: USER PROBES



User-Defined Probes:

Usage:

```
sigmalnst -p event_description probelib.a appbin  
./appbin.inst
```

- Allows the user to inject code (`probelib.a`) into a binary application.
- The `event_description` file specifies which event should trigger each function in `probelib.a`
- The following events are defined:
 - **Load** myfun(): call myfun() each time a load operation is executed
 - **Store** myfun(): call myfun() each time a store operation is executed
 - **Malloc** myfun(): call myfun() each time the malloc/allocate function is invoked
 - **Free** myfun(): call myfun() each time the free/deallocate function is invoked
 - **F_entry** f() myfun(): call myfun() each time function f() is entered
 - **F_exit** f() myfun(): call myfun() each time function f() is exited
 - **F_replace** f() myfun(): completely replace the function f() with myfun()
 - **Cachesim**: activate the cache simulator
 - **Symbmap**: activate the symbolic mapping

User-Defined Probes:

- The probe functions have access to different information, depending on the event that triggered the probe:
 - **Load, Store, Malloc, Free** events:
 - **Instruction** that triggered the event
 - **Virtual address** of the instruction that triggered the event
 - **FileName, FunctionName, lineNumber** of the instruction that triggered the event
 - **Load and Store** events:
 - **Address** loaded or stored
 - **Current stack pointer**
 - **Data** loaded or stored
 - **Malloc and Free** events:
 - **Address** allocated or freed
 - **Number of bytes** allocated
 - **F_entry, F_replace** events:
 - The probe function is invoked with the same arguments as the intercepted function

User-Defined Probes:

- When the **cachesim** event is defined, the probe functions triggered by load and store events have also access to the details of the cache simulation:
 - Whether the load or store that triggered the event was a **Hit**, a **Miss** or a **Cold Miss** in each cache level and in the TLB
 - Which blocks were **evicted** in each cache level (if any) as a result of the load/store
 - How many hits and misses occurred as a **side effect** of the current load (prefetching requests, evictions etc)
 - Which blocks were **evicted** in each cache level as a side effect of the load/store
 - Furthermore, the **MD memory profiles** are also generated at the end of the run
- When the **symbmap** event is defined, the probe functions triggered by load and store events have also access to
 - Which **symbol name** does the loaded/stored address correspond to
 - The symbolic mapping supports: **global variables**, **stack variables** and **dynamically allocated variables**
- The cachesim event includes the symbmap event automatically

User-Defined Probes:

- We are extending the infrastructure to
 - Allow **symbolic specification of events**
 - Invoke the probe when “A” is touched
 - Invoke the probe when “A” is loaded from function “f”
 - etc...
 - Allow specification of **events based on performance metrics**
 - Invoke the probe each time “A” has an L1 miss
 - Invoke the probe if “A” has more than 10% load misses in function “f”
 - etc...
 - Provide a simple **grammar** and a **GUI** to select the events and instrument the application

The MD Backend:

Usage:

```
sigmalnst -d appbin  
./appbin.inst -sigma archFile
```

- Execute detailed cache simulation and provide memory profile
- Power3/Power4 architectures prefetching implemented
- Write-Back/Write-Through caches, replacement policies etc

archFile specifies:

- Number of cache levels
- Cache size, line size, associativity, repl. policy, write policy, prefetching algorithm
- Multiple machines can be simulated at once

The MD Backend: memory profile

Provide counters such as hits, misses, cold misses for

each cache level

each function

each data structure

each data structure within each function

– Output sorted by the SIGMA **memtime**:

**SUM(LoadHits(i)*LoadLat(i) + StoreHits(i)*StoreLat(i)) +
#TLBmisses * Lat(TLBmiss)**

memtime should track wall time for memory bound applications

Memory Profile Output

	L1	L2	L3	TLB	MEM
FUNCTION: calc1 (memtime = 0.0050)					
Load Acc/Miss/Cold	522819/2252/1	2252/345/0	345/0/0	784419/126/0	0/-/0
Load Acc/Miss Ratio	232	6	-	6225	-
Load Hit Ratio	99.57%	84.68%	100.00%	99.98%	-
Est. Load Latency	0.0008 sec	0.0000 sec	0.0000 sec	0.0001 sec	0.0000 sec
Load Traffic	-	238.38 Kb	43.12 Kb	-	0.00 Kb
.....					
FUNCTION: calc2 (memtime = 0.0042)					
Load Acc/Miss/Cold	622230/2631/0	2631/1661/0	1661/0/0	814269/94/0	0/-/0
Load Acc/Miss Ratio	236	1	-	8662	-
Load Hit Ratio	99.58%	36.87%	100.00%	99.99%	-
Est. Load Latency	0.0010 sec	0.0000 sec	0.0001 sec	0.0001 sec	0.0000 sec
Load Traffic	-	121.25 Kb	207.62 Kb	-	0.00 Kb
.....					
	L1	L2	L3	TLB	MEM
DATA: u (memtime = 0.0012)					
Load Acc/Miss/Cold	167710/708/0	708/317/0	317/0/0	216097/31/0	0/-/0
Load Acc/Miss Ratio	236	2	-	6970	-
Load Hit Ratio	99.58%	55.23%	100.00%	99.99%	-
Est. Load Latency	0.0003 sec	0.0000 sec	0.0000 sec	0.0000 sec	0.0000 sec
Load Traffic	-	48.88 Kb	39.62 Kb	-	0.00 Kb
.....					
DATA: v (memtime = 0.0012)					
Load Acc/Miss/Cold	167710/721/0	721/316/0	316/0/0	216097/31/0	0/-/0
Load Acc/Miss Ratio	232	2	-	6970	-
Load Hit Ratio	99.57%	56.17%	100.00%	99.99%	-
Est. Load Latency	0.0003 sec	0.0000 sec	0.0000 sec	0.0000 sec	0.0000 sec
Load Traffic	-	50.62 Kb	39.50 Kb	-	0.00 Kb
.....					

Memory Profile Viewer – Data Structure Focus

peekperf

File Tools

sigma

Label	MemTime /	Access
u@Global	1.57702e+06	216097
u@Global_inital	447628	32511
<u>u@Global_calc3</u>	<u>444718</u>	<u>32258</u>
u@Global_calc3z	433822	32258
u@Global_calc1	250850	119070
v@Global	1.57569e+06	216097
p@Global	1.52905e+06	192786
h@Global	1.30777e+06	168216
z@Global	1.30759e+06	168216
pold@Global	1.27448e+06	112144
vold@Global	1.27217e+06	112144
uold@Global	1.27047e+06	112144
cv@Global	1.26258e+06	145158
cu@Global	1.26066e+06	145158
unew@Global	1.19097e+06	81151
vnew@Global	1.15966e+06	81151
pnew@Global	1.15684e+06	81151
psi@Global	486142	51913
unknownDt@Global	11220	539
unknownDt@Local	424	30

swim.f

```

VOLD(N1,N2), POLD(N1,N2),
2  CU(N1,N2), CV(N1,N2),
*  Z(N1,N2), H(N1,N2), PSI(N1,N2)
C
COMMON /CONS/ DT, TDT, DX, DY, A, ALPHA, ITMAX, MPRINT, M, N, MP1,
1  NP1, EL, PI, TPI, DI, DJ, PCF
C
TDT = TDT+TDT
DO 400 J=1, NP1
DO 400 I=1, MP1
UOLD(I, J) = U(I, J)
VOLD(I, J) = V(I, J)
POLD(I, J) = P(I, J)
U(I, J) = UNEW(I, J)
V(I, J) = VNEW(I, J)
P(I, J) = PNEW(I, J)
400 CONTINUE
RETURN
END
C SPEC removed CCMIC$ MICRO
SUBROUTINE CALC3
C
C TIME SMOOTHER
C
IMPLICIT REAL*8 (A-H, O-Z)
#include "swim.h"
COMMON U(N1,N2), V(N1,N2), P(N1,N2)
                
```

Metric Browser: u@Global_calc3

Task	thread	Misses	LoadMisses	StoreMisses	Hits	LoadHits	StoreHits	Access	LoadAccesses	StoreAccess
0	L1	442	193	249	31816	15936	15880	32258	16129	16129
0	L2	190	74	116	16132	119	16013	16322	193	16129
0	L3	0	0	0	1191	74	1117	1191	74	1117
0	TLB	0	0	0	32258	32258	0	32258	32258	0

C SPEC removed CCMIC\$ DO GLOBAL

19

SCICOMP 2004

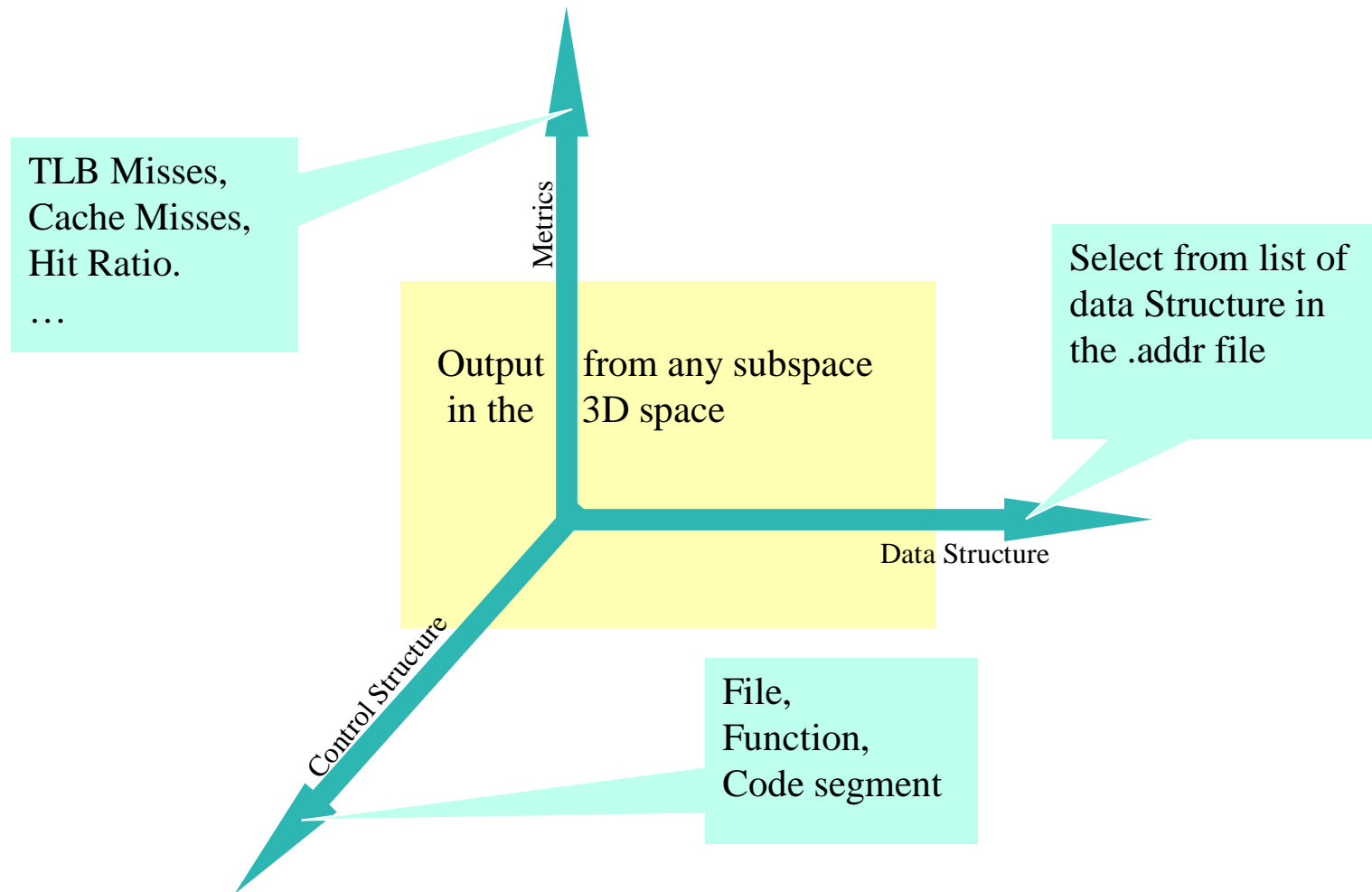
SIGMA: A Software Infrastructure to Guide Memory Analysis

Memory Profile Viewer – Function Focus

The screenshot shows the Memory Profile Viewer (peekperf) interface. The main window is divided into two panes. The left pane shows a tree view of memory accesses, with 'calc1_cv@Global' selected. The right pane shows the source code for 'swim.f', which includes a subroutine 'CALC1' that computes capital U, V, Z, and H. A 'Metric Browser' window is open for 'calc1_cv@Global', displaying a table of performance metrics.

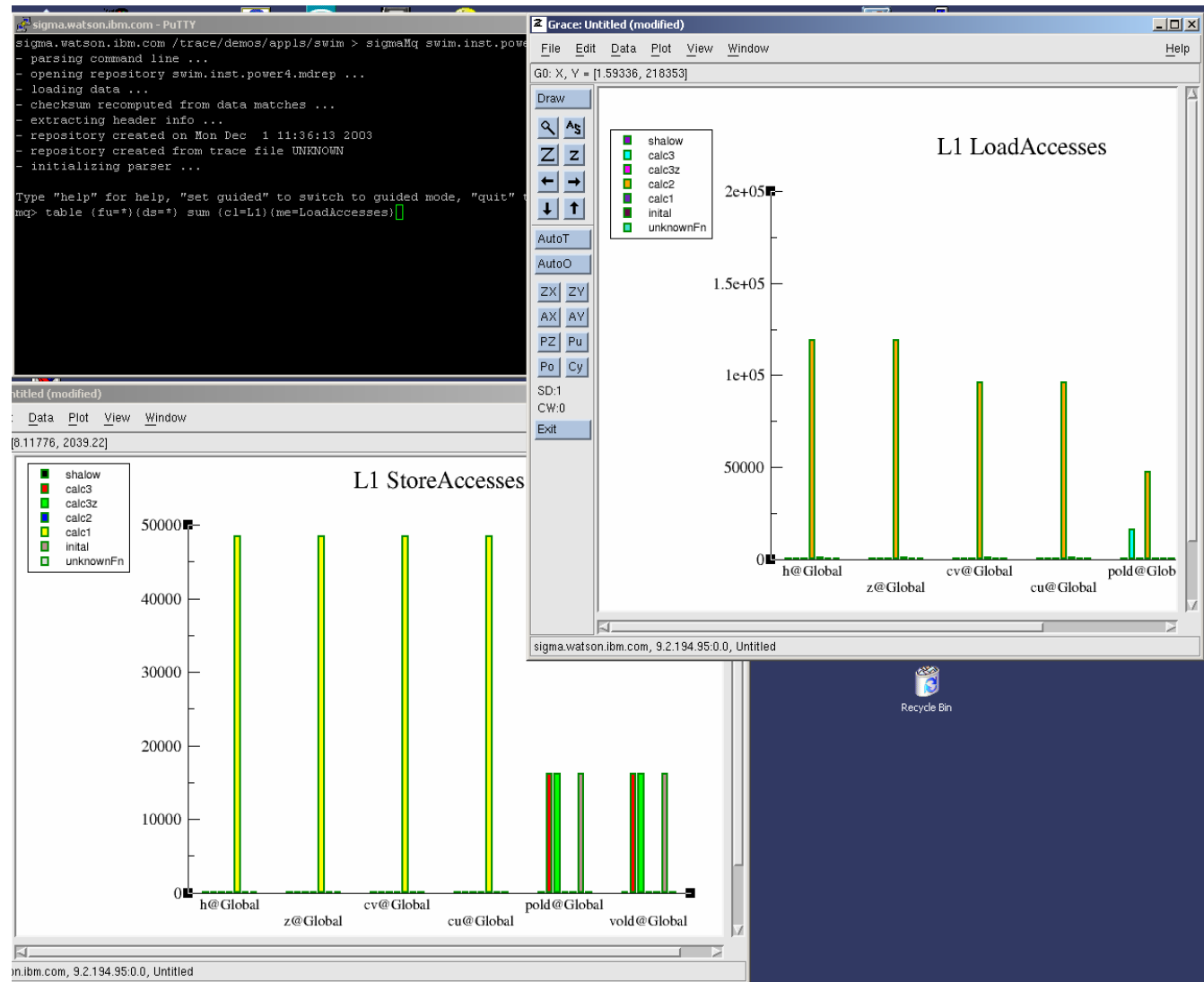
sses	StoreAccesses	LoadMissesCold	StoreMissesCold	MemTime	ConflictPressure	CapacityPressure	ColdPressure
48387	0	0	1009	1.05255e+06	0.750824	99.2492	1.00757
48387	0	0	1009	0	93.7942	6.20579	16.114
3027	0	0	253	0	91.6419	8.35811	11.9644
0	32	32	0	0	99.9349	0.065112	1535.81

SIGMA Repository



Visualiz: Query Language

- Build tables and lists
- ASCII and bar-chart output
- Support arithmetic operators (+ - * / ./)
- Compute derived Metrics



Example: Performance Modeling with SIGMA

- We started with the simplest performance model
let A be a memory architecture and S a problem size

$$mt(A,S) = \text{SUM}(\text{LoadHits}(i)*\text{LoadLat}(i) + \text{StoreHits}(i)*\text{StoreLat}(i)) + \text{TLBmisses} * \text{Lat}(\text{TLBmiss})$$

$mt(A,S)$ is very crude and cannot closely represent $wt(A,S)$,
however what we are really interested in is the trend of the ratio

$$PR(A,S) := wt(A,S) / mt(A,S)$$

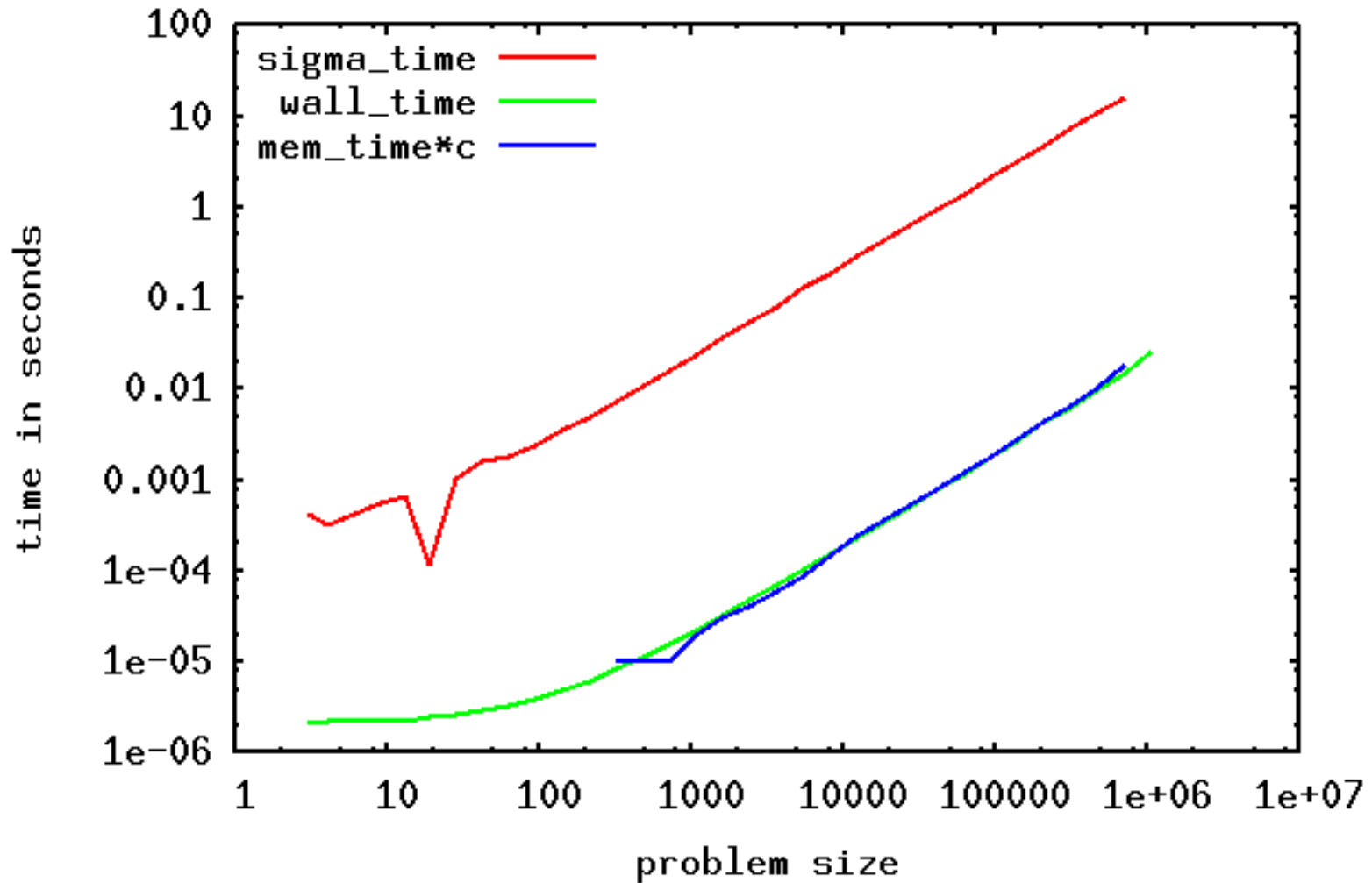
- Is the function $F(S) = PR(A,S)$ constant? (**MemScalPrediction**)
- Is the function $G(A) = PR(A,S)$ constant? (**MemPerfPrediction**)

MemScalPrediction:

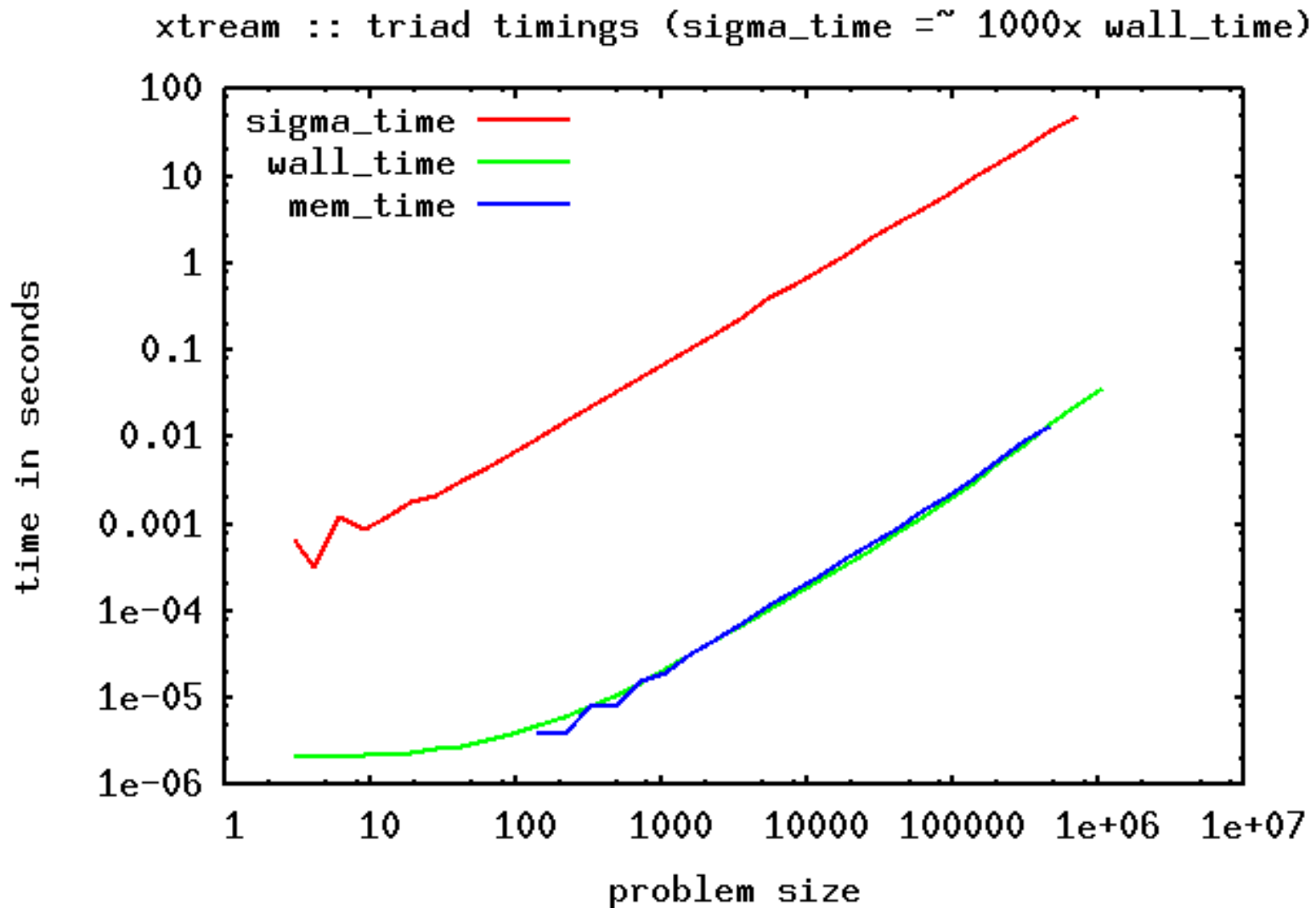
- The function $F(S) = wt(A,S) / mt(A,S)$ is generally constant if
 - **The application is memory-intensive, such that compute time is negligible w.r.t. memory time**
 - **Concurrency within mem ops scales with size and hence can be ignored**
- Initial testing with xstream and NAS benchmarks shows good agreement between wt and mt (NERSC, May 2004)

Fill: $A[i] = s$ walltime is proportional to memtime

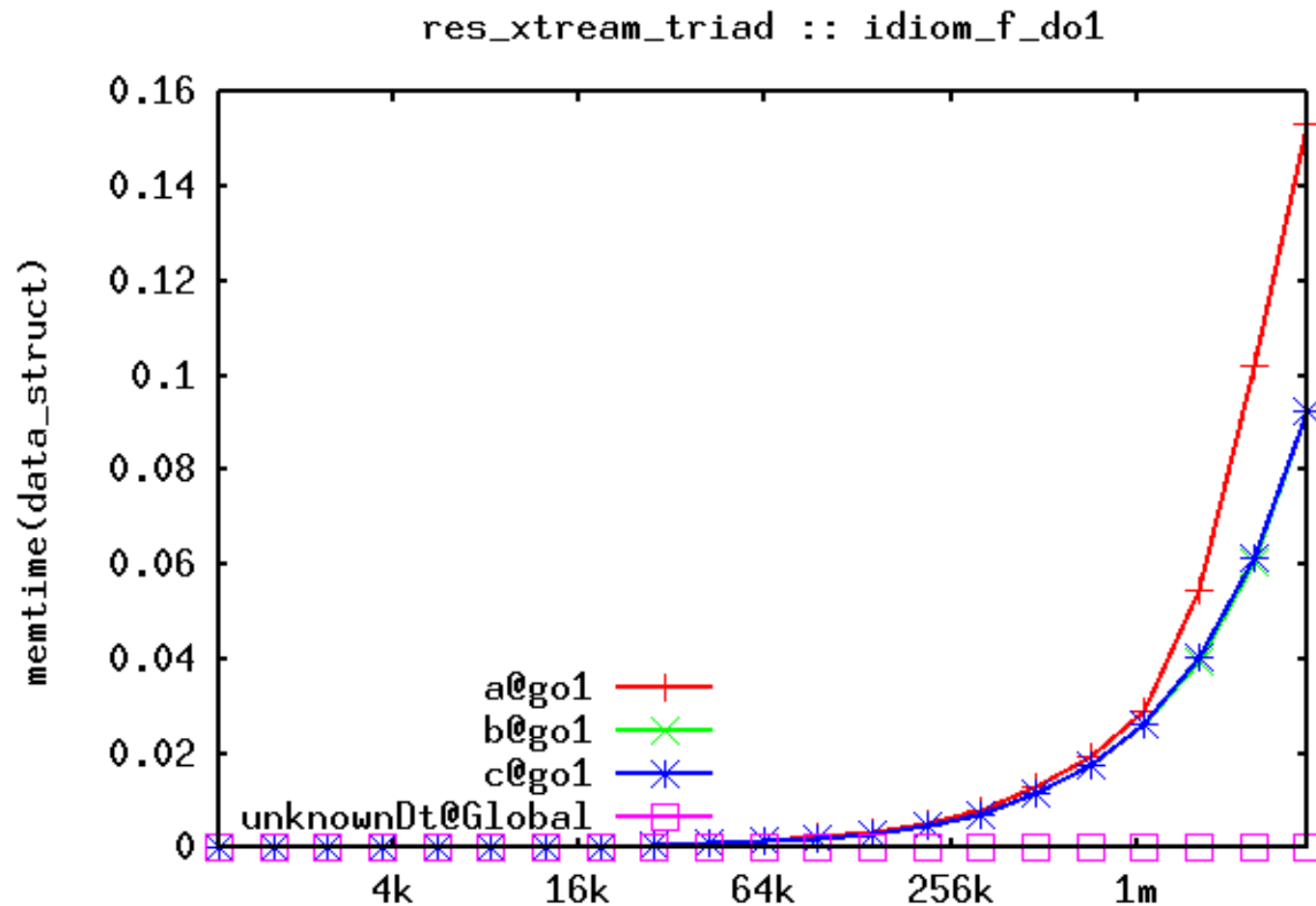
xstream :: fill timings (sigma_time \approx 1000x wall_time)



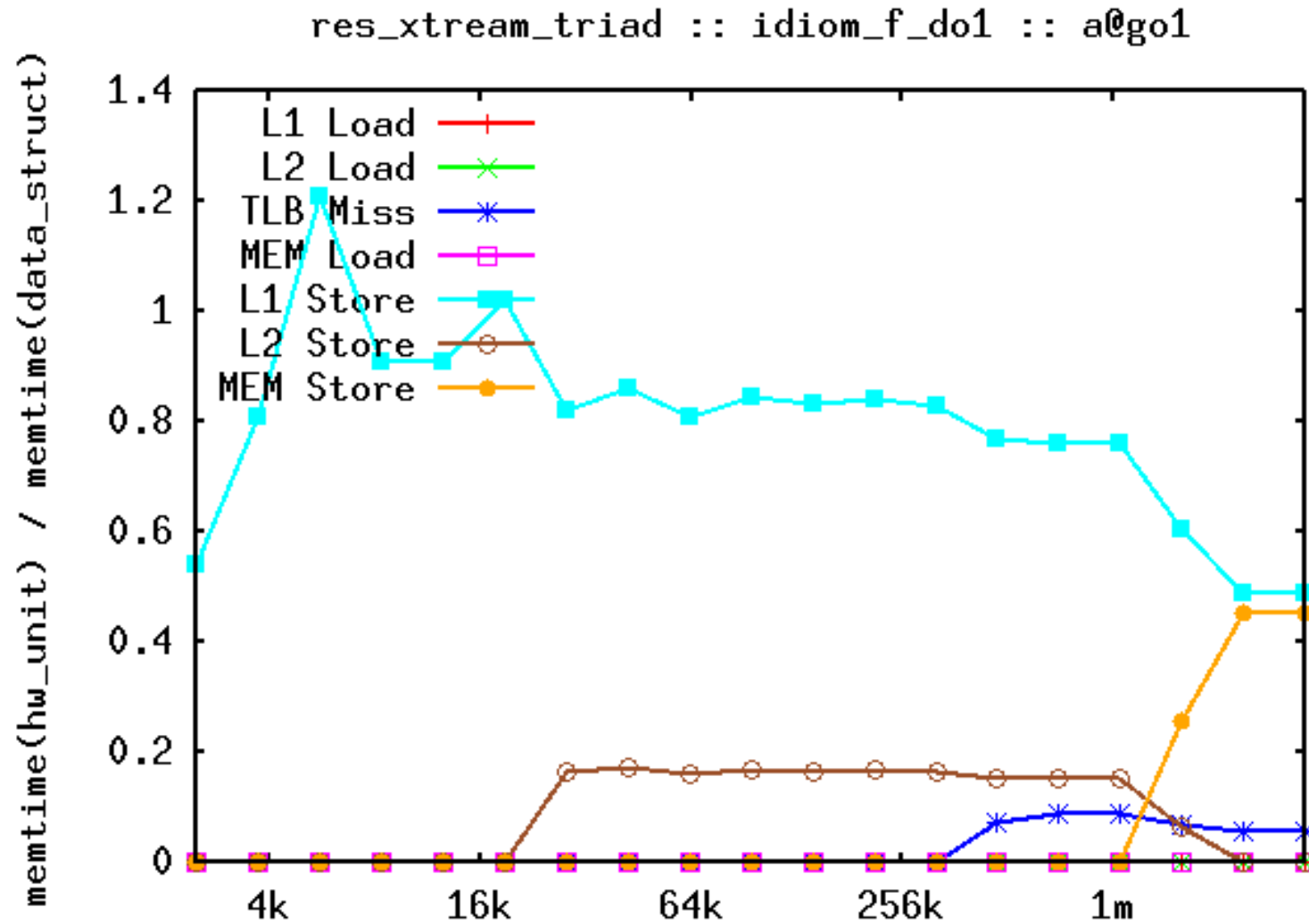
Triad: $A[i] = B[i] + sC[i]$ walltime is proportional to memtime



Triad: $a[i] = b[i] + s * c[i]$ memtime for a,b,c

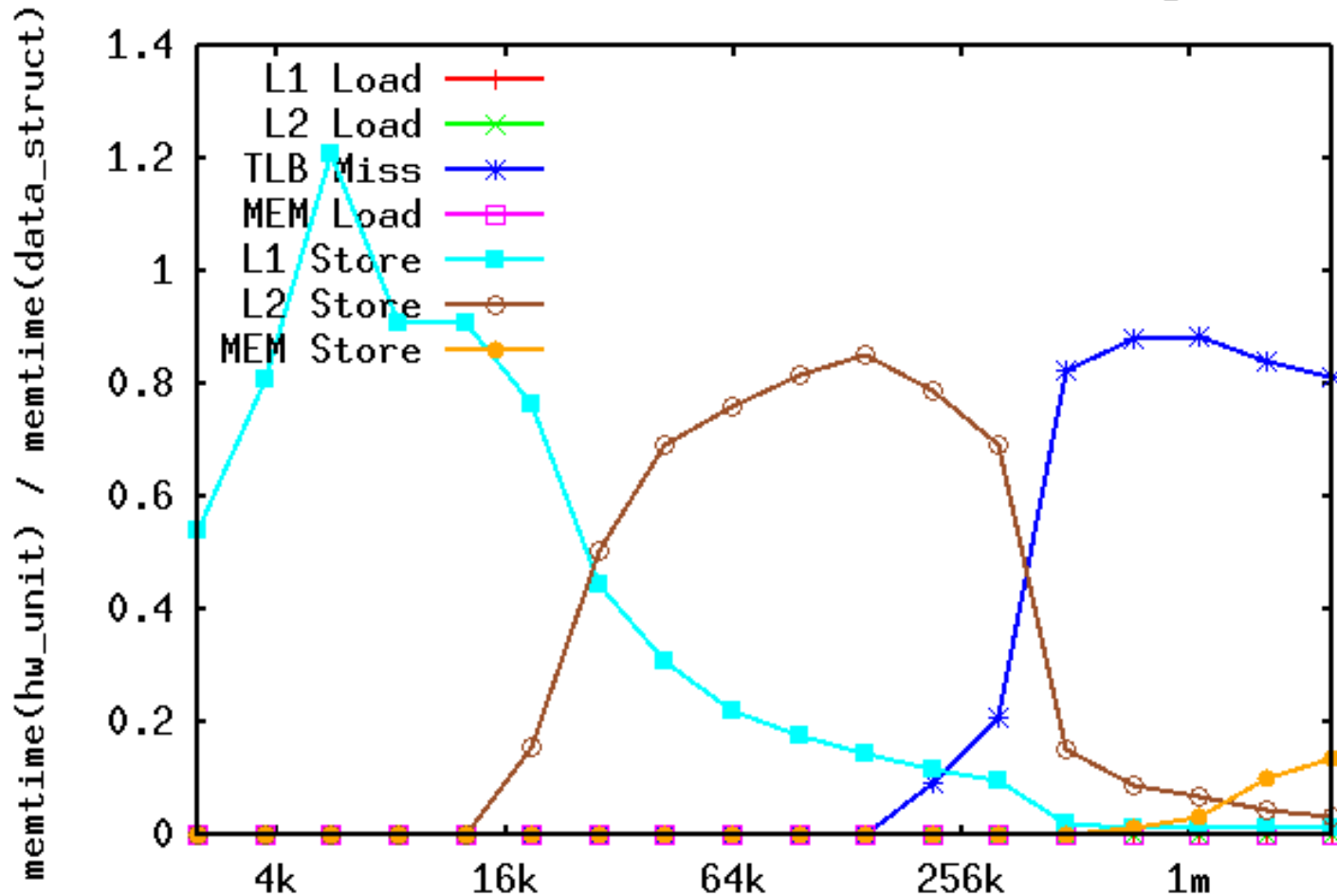


Triad: $a[i] = b[i] + s * c[i]$, memtime per hw unit (a)



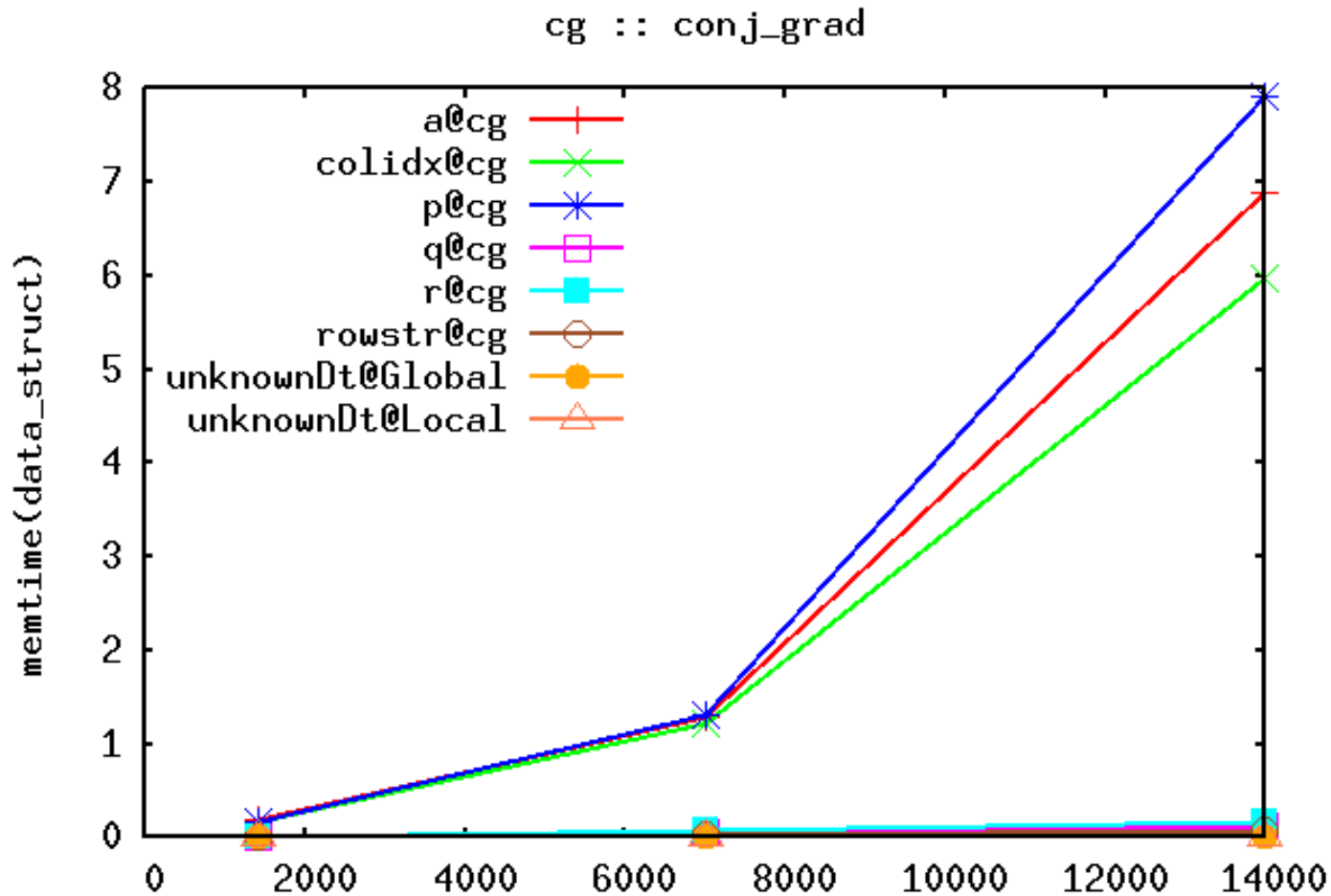
iTriad: $a[ia[i]] = b[ib[i]] + s*c[ic[i]]$, memtime per hw unit (a)

```
res_xstream_itriad :: idiom_f_do1 :: a@go1
```

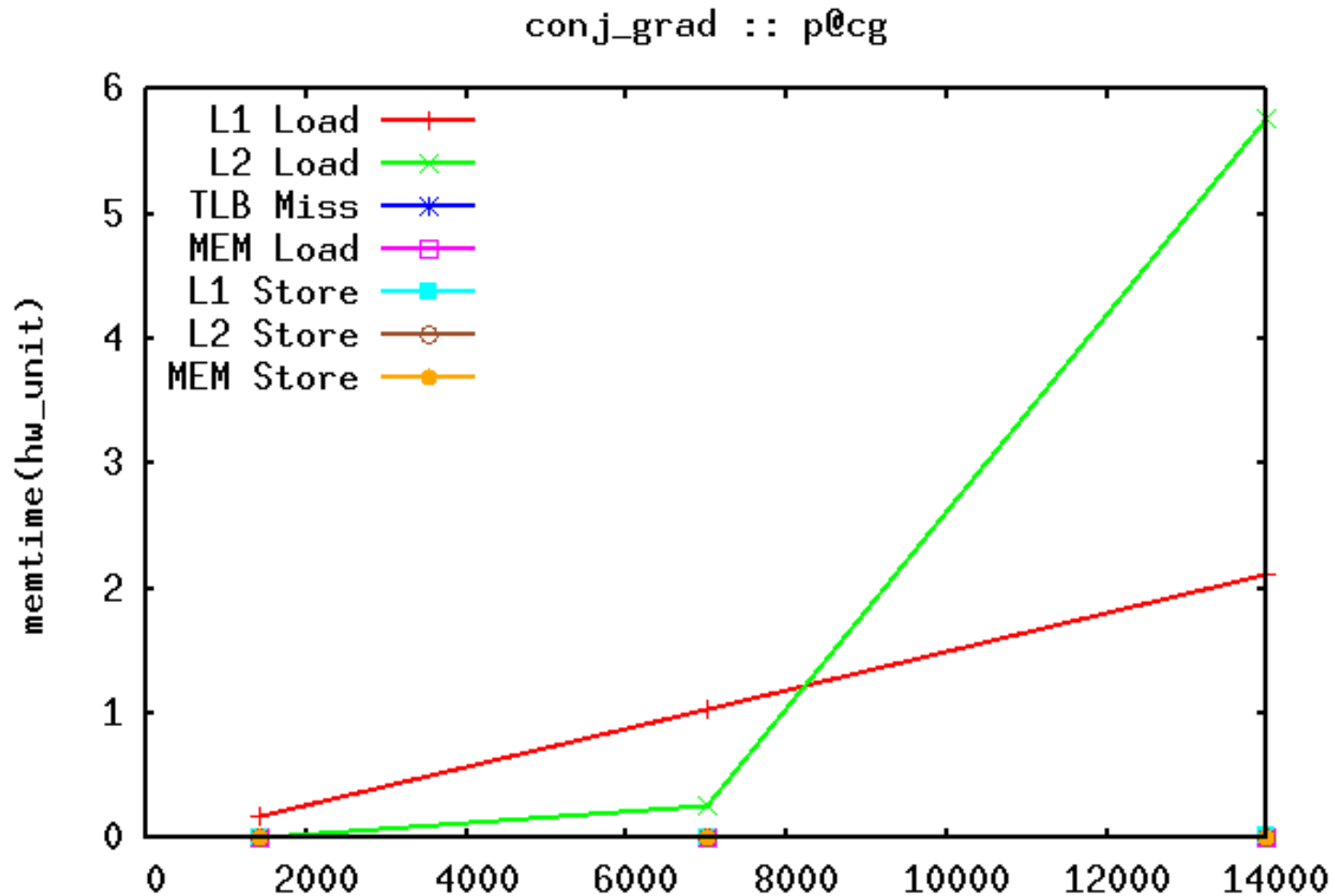


Excercising all the components of the memory subsystem.

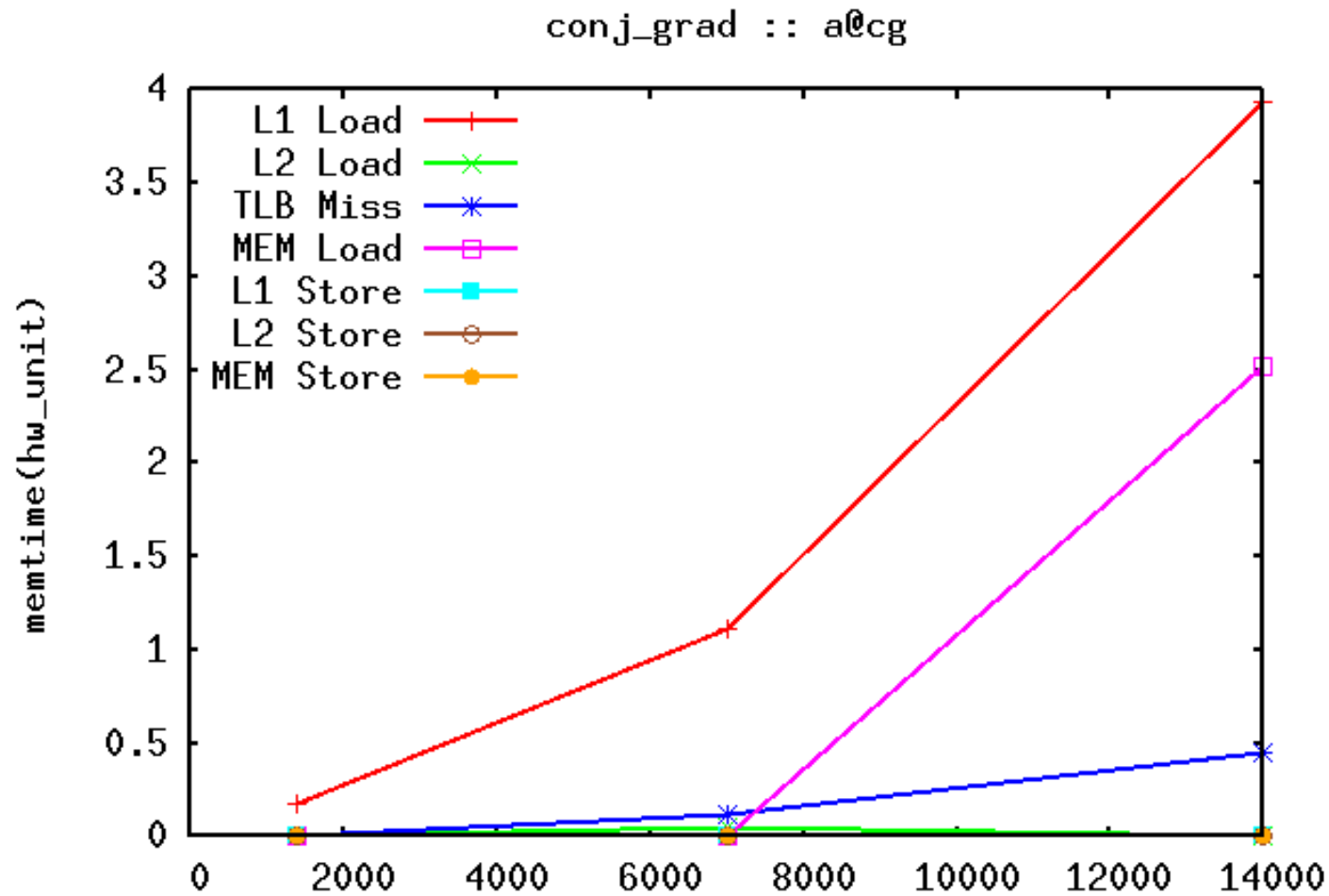
NPB CG: a, p and colidx dominate the memtime



NPB CG: memtime per hw unit (p)



NPB CG: memtime per hw unit (a)



NPB CG: relating metrics to program source

```
! q = A.p
do j=1,lastrow-firstrow+1
    sum = 0.d0
    do k=rowstr(j),rowstr(j+1)-1
        sum = sum + a(k)*p(colidx(k))
    enddo
    w(j) = sum
enddo
```

The TR Backend:

Usage:

```
sigmaInst -d -dback tr appbin  
./appbin.inst
```

- Generate a compressed memory trace
- All the backends can run on the compressed trace file:

```
sigmaMd tracefile -sigma archFile  
sigmaFs tracefile
```

- Control-Flow based Trace Compression (Patented):
 - Compress the addresses produced by each instruction separately
 - Capture strides, repetitions, nested patterns
 - Compression is performed online
 - Compress all trace events, not only addresses

SIGMA Trace Compression Rate

	# Mem References	Uncompressed Trace	Compressed Trace	Compression Rate
NAS bt.A	13,279,268,500	61.8 Gb	325 Mb	194.71
NAS cg.A	1,735,939,092	8.8 Gb	3.52 Gb	2.5
NAS ep.A	4,698,767,861	22.3 Gb	2.56 Gb	8.71
NAS ft.A	8,576,076,276	40 Gb	870 Mb	47.08
NAS is.A	1,325,400,450	1.32 Gb	540 Mb	2.5
NAS lu.A	7,819,640,763	36.42 Gb	37 Kb	1,024
NAS mg.A	6,309,492,654	29.4 Gb	17 Mb	1770.91
NAS sp.A	8,986,253,493	41.8 Gb	323 Mb	132.51
swim 1023	129,171,002	615 Mb	8.3 Kb	75874
array1	45,650,677	217 Mb	6 Kb	37034
mmblock	6,500,000	29.7 Mb	0.0014 Mb	21,214.29
Bitonic sort	16,300,000	100 Mb	66.5 Mb	1.50
Rec. mmult	18,900,000	102.1 Mb	101.1 Mb	1.01
Red black SOR	137,100,000	574.3 Mb	14.6 Mb	39.34
SPEC applu	59,500,000	243.6 Mb	0.17 Mb	1,432.94
SPEC hydro2d	130,300,000	643.8 Mb	0.83 Mb	775.66
SPEC mgrid	100,200,000	415 Mb	0.4 Mb	1,037.50
SPEC swim	147,900,000	685.6 Mb	0.011 Mb	62,327.27
SPEC wupwise	375,900,000	2.6 Gb	477.6 Mb	5.57
Average	2,836,750,567	13.05 Gb	471.69 Mb	28.3

The FS Backend (work in progress):

Usage:

```
sigmalnst -d -dback fs -domp appbin  
./appbin.inst
```

Target: memory performance of shared-memory apps (infinite cache)

- Detect cache misses due to invalidation
- Detect false-sharing misses
- Provide suggestions for code rearrangement to minimize false-sharing misses
 - **Remapping of data structures**
 - **Remapping of computation**
 - **Changing scheduling policy**

The FS Backend (work in progress):

Example: invalidation misses in NAS CG.A: p is responsible for 86% of all cache invalidations

Array	Function	Mem References	Stores	Cold Misses	Cache Invalidation Misses
p	conj_grad@OL@B@OL@F	741241600	0	2624	1048576
a	conj_grad@OL@B@OL@F	741241600	0	16588	70057
z	conj_grad@OL@C@OL@13	29649664	0	2621	39427
z	conj_grad@OL@B@OL@11	11200000	5600000	0	2395
r	conj_grad@OL@B@OL@11	11200000	5600000	0	1601
p	conj_grad@OL@B@OL@12	11200000	5600000	0	1600
q	conj_grad@OL@B@OL@F	5600000	5600000	0	1271
q	conj_grad@OL@B@OL@10	5600000	0	0	1200
r	conj_grad@OL@B@OL@12	5600000	0	0	1200
colidx	conj_grad@OL@B@OL@F	741241600	0	0	403
q	conj_grad@OL@B@OL@11	5600000	0	0	389
rowstr	cg@OL@1@OL@7	42000	0	0	331
p	conj_grad@OL@A@OL@D	224016	224016	0	105
z	conj_grad@OL@A@OL@D	224016	224016	0	69
q	conj_grad@OL@A@OL@D	224016	224016	0	67
r	conj_grad@OL@A@OL@E	224000	0	0	50
x	conj_grad@OL@A@OL@D	224016	0	0	48
r	conj_grad@OL@C@OL@13	224000	224000	0	48
r	conj_grad@OL@C@OL@14	224000	0	0	48
r	conj_grad@OL@A@OL@D	224016	224016	0	47
x	cg@OL@3	210000	210000	0	45
colidx	conj_grad@OL@C@OL@13	29649664	0	0	16
x	conj_grad@OL@C@OL@14	224000	0	1	15

The FS Backend (work in progress):

Example: invalidation misses in NAS CG.A:

```
do iter=1,maxiter

  !$omp do
  do j=1,naa+1
    q(j) = 0.0d0
    z(j) = 0.0d0
    r(j) = x(j)
    p(j) = r(j)
  enddo

  !$omp do
  do j=1,lastrow-firstrow+1
    sum = 0.d0
    do k=rowstr(j),rowstr(j+1)-1
      sum = sum + a(k)*p(colidx(k))
    enddo
    q(j) = sum
  enddo

enddo
```

Next Steps

- Complete the support for shared-memory apps
 - Cache simulation for multi-threaded applications
 - Which interleaving for parallel threads?
- Expose the prefetching API
- Extend the event selection to include symbolic and performance related events
- Support 64-bit applications
- Improve the performance of the cache simulator (sampling?)
- Performance projection



Advanced Computing Technology Center

Questions / Comments