Commercial System Performance Analysis at Sun

Brian O'Krafka

Sun Labs Performance and Architecture Group

Commercial System Performance Analysis at Sun

- Introduction
- System Modeling
  - Typical System Model
  - Modeling Environment
- Workload characterization
  - Miss rate analysis
  - Processor abstraction using blocking factors

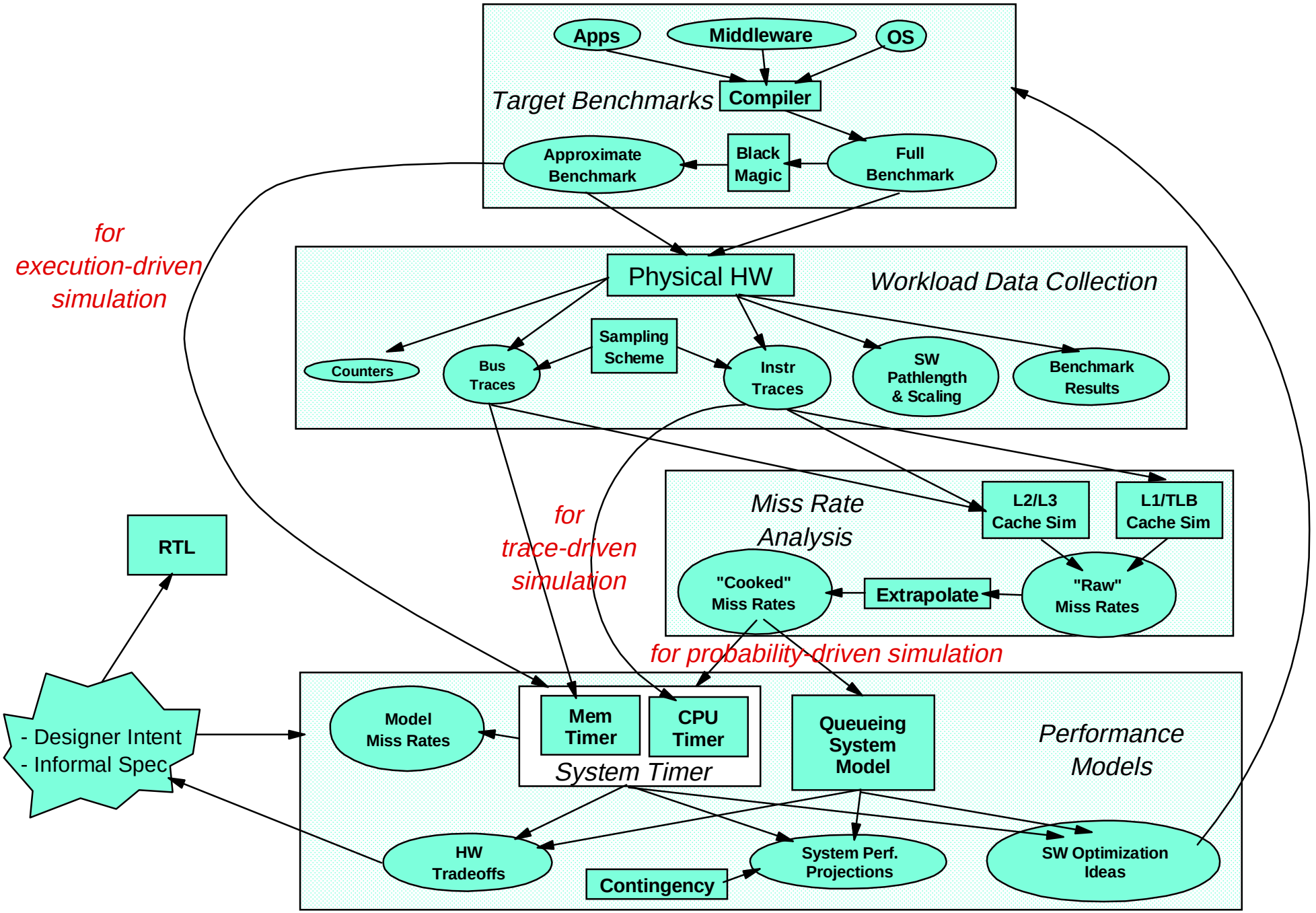
The Problem

- Timely, accurate performance projections for Sun systems
- Why?
  - early exploration of broad design space
  - engineering tradeoffs (alternative processors, cache sizes, interconnect topology, bus sizing, ...)
  - validation of more detailed models
  - portfolio management (business planning)
- Scope:
  - single cache coherent multiprocessor systems
  - "well-behaved" commercial workloads

Constraints

- workloads are a moving target
- traces are very hard to get (esp. instruction traces)
- never have traces for all configurations of interest
- need broad design space exploration early in project
- need good accuracy
- detailed, low-level system models aren't a complete solution:
  - detail is unavailable early in project
  - don't have traces to drive configurations of interest
  - difficult to write and debug
- many, many configurations

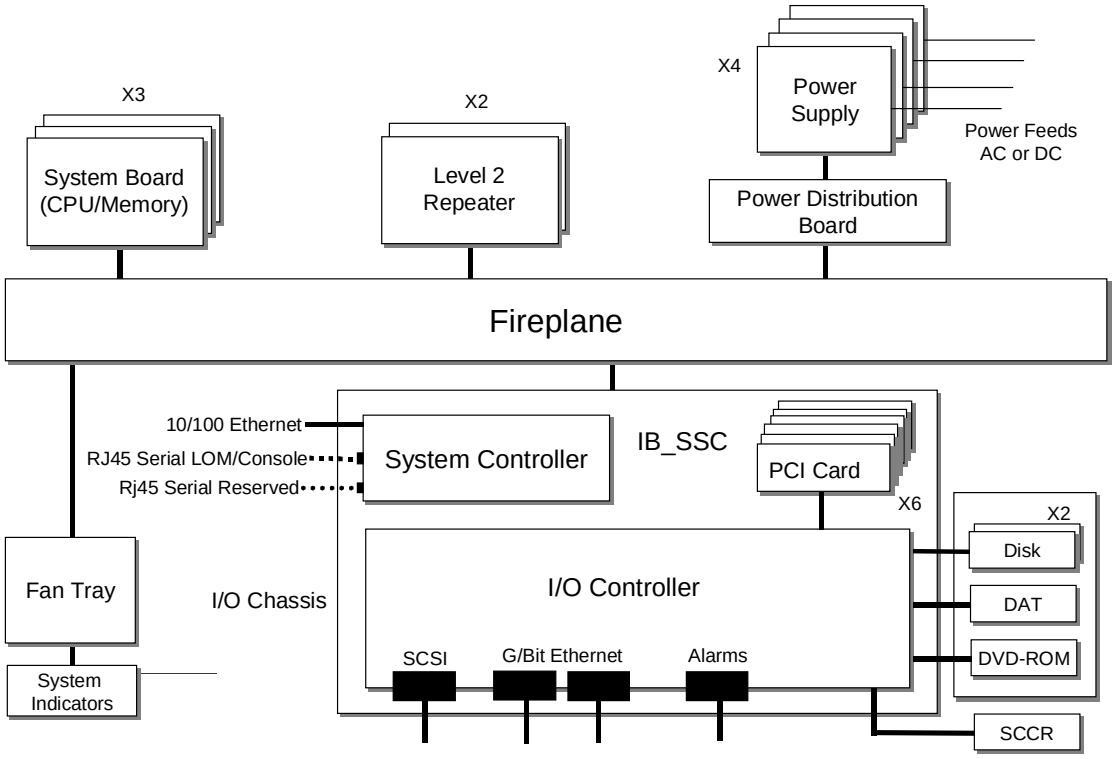
Performance Analysis Process



System Modeling:

- Typical Model
- Sysmodel Environment

Figure 3-1: Architecture of the Sun Fire V1280 Server



The SPARC Microprocessor Design

The Version 9 Architecture

The SPARC architecture has been implemented in processors used in a range of systems from laptops to supercomputers. SPARC International member companies have implemented numerous compatible microprocessors since the SPARC platform was first announced — more than any other RISC (reduced instruction set computing) microprocessor family. As a result, the SPARC architecture boasts the support of thousands of compatible software and hardware products. SPARC Version 9 maintains upwards binary compatibility for application software developed for previous SPARC architecture implementations, including microSPARC®, TurboSPARC®, SuperSPARC®, and previous versions of UltraSPARC.

The SPARC V9 architecture represents a significant advance for the microprocessor industry. It provides 64-bit data and addressing, fault tolerance features, fast context switching, support for advanced compiler optimizations, efficient design for superscalar processors, and a clean structure for emerging operating systems. And all of this has been accomplished with 100-percent binary compatibility for existing applications.

The UltraSPARC III Cu Processor

The UltraSPARC III Cu processor is part of a third generation of UltraSPARC pipeline-based products. In addition to using a new process technology, the UltraSPARC III Cu processor provides a higher clock frequency, reduced on-chip latencies, support for greater amounts of level-one and level-two cache, and an integrated external memory controller. Other features include support for

a critical design point since misses from large caches tend to cluster, with adjacent misses impacting performance in systems without high memory transfer rates.

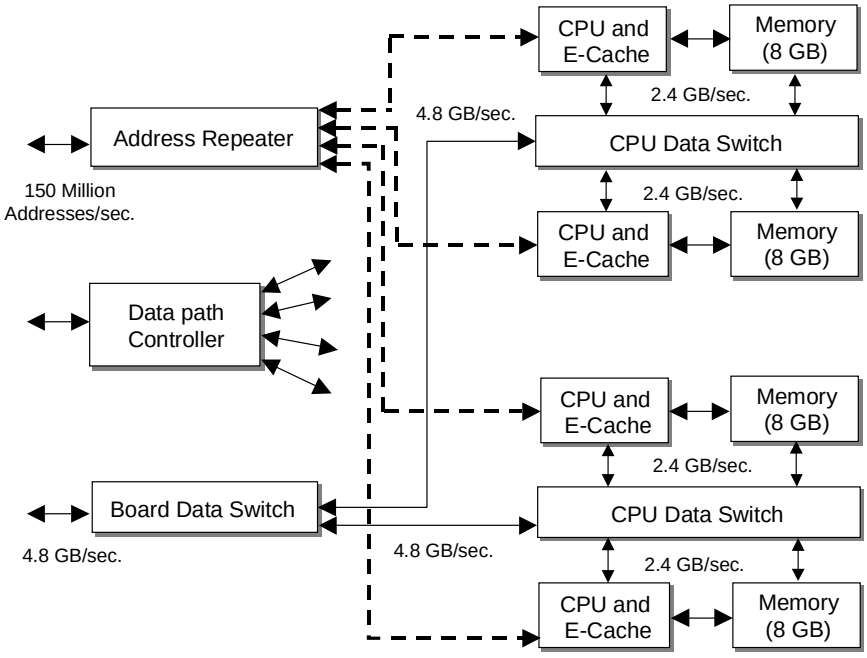
- *System Interface Unit (SIU)*
  Charged with the task of all other off-processor communications (memory, other processors, and I/O devices) the SIU can handle up to 15 pending transactions with support for full out-of-order data delivery on each transaction, enabling memory banks in a MP system service a request as soon as a bank is available. All processor interfaces use error detection and/or correction codes to quickly detect errors. In the event of an error on the system bus, an independent 8-bit-wide back door bus allows the use of automated diagnostics to isolate the problem.

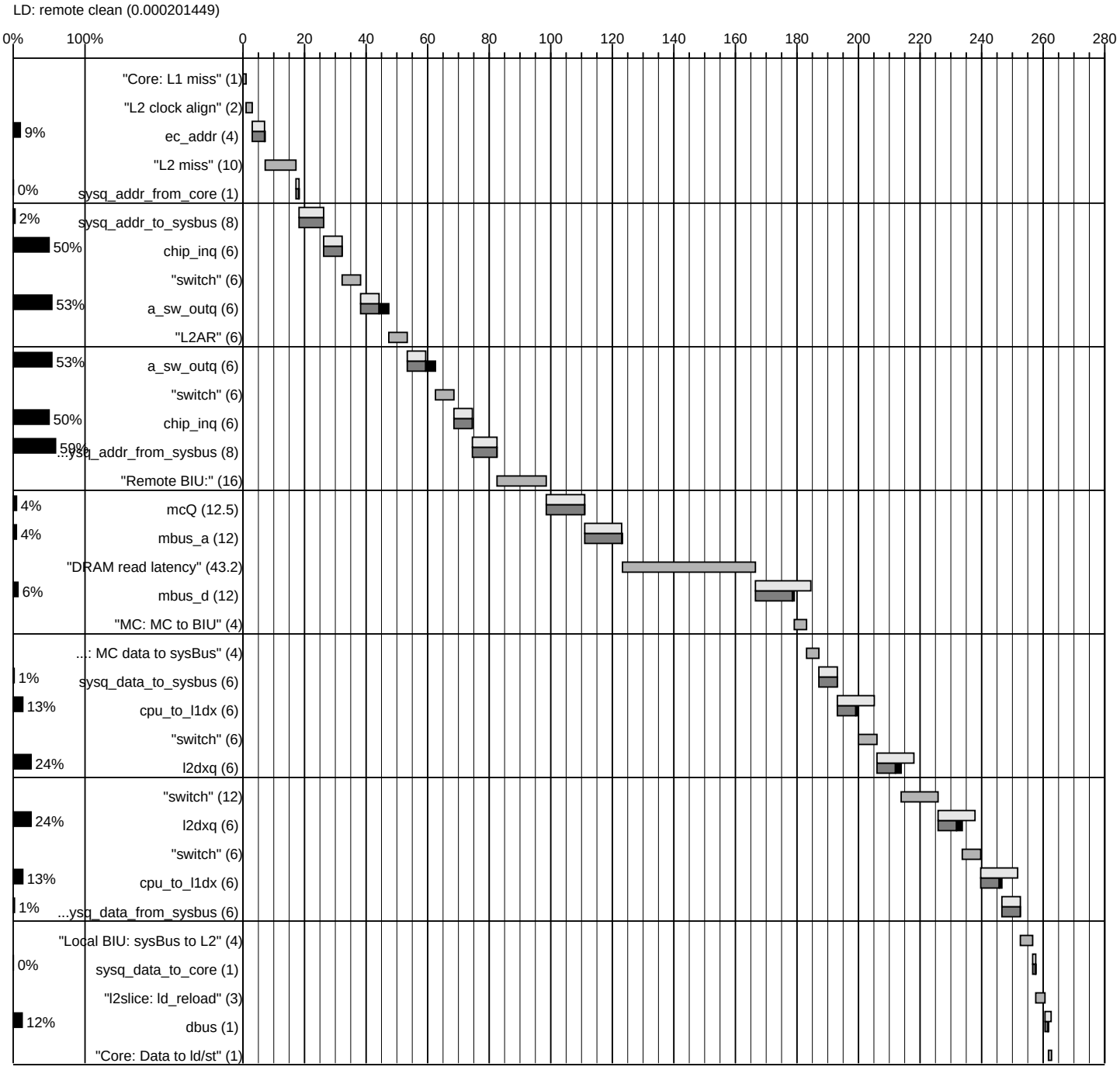
CPU/Memory Boards

The Sun Fire V1280 server can accommodate up to twelve UltraSPARC III Cu processors populated on three CPU/memory boards. Each board includes four processors, all cache, and main memory. Memory can be added to a board after initial installation by trained service personnel. In addition, while all of the processors on a single CPU/memory board must be the same speed, other CPU/memory boards within the system may use processors clocked at a different speed. This mixed-speed CPU support results in better investment protection when upgrading by precluding the need to replace all of the existing processors in a system.

The block diagram of the CPU/memory board used in the Sun Fire V1280 server is shown in Figure 3-4. Address and control paths are illustrated with dashed lines, and data path with solid lines. The interconnect components on the left connect to the Sun Fireplane interconnect switch boards. The bandwidths shown are the peak at each point on the board. (Electronically, the CPU/memory boards are identical to the Uniboard used in the Sun Fire 3800-6800 servers. However, they are physically incompatible and are not interchangeable.)

Figure 3-4: CPU/Memory Board Block Diagram




LD: remote clean (0.000201449)
0%
100%
0
20
40
60
80
100
120
140
160
180
200
220
240
260
280
"Core: L1 miss" (1)
"L2 clock align" (2)
9%
ec_addr (4)
"L2 miss" (10)
0%
sysq_addr_from_core (1)
2%
sysq_addr_to_sysbus (8)
50%
chip_inq (6)
"switch" (6)
53%
a_sw_outq (6)
"L2AR" (6)
53%
a_sw_outq (6)
"switch" (6)
50%
chip_inq (6)
50%
...ysq_addr_from_sysbus (8)
"Remote BIU:" (16)
4%
mcQ (12.5)
4%
mbus_a (12)
"DRAM read latency" (43.2)
6%
mbus_d (12)
"MC: MC to BIU" (4)
...: MC data to sysBus" (4)
1%
sysq_data_to_sysbus (6)
13%
cpu_to_l1dx (6)
"switch" (6)
24%
l2dxq (6)
"switch" (12)
24%
l2dxq (6)
"switch" (6)
13%
cpu_to_l1dx (6)
1%
...ysq_data_from_sysbus (6)
"Local BIU: sysBus to L2" (4)
0%
sysq_data_to_core (1)
"l2slice: ld_reload" (3)
12%
dbus (1)
"Core: Data to ld/st" (1)

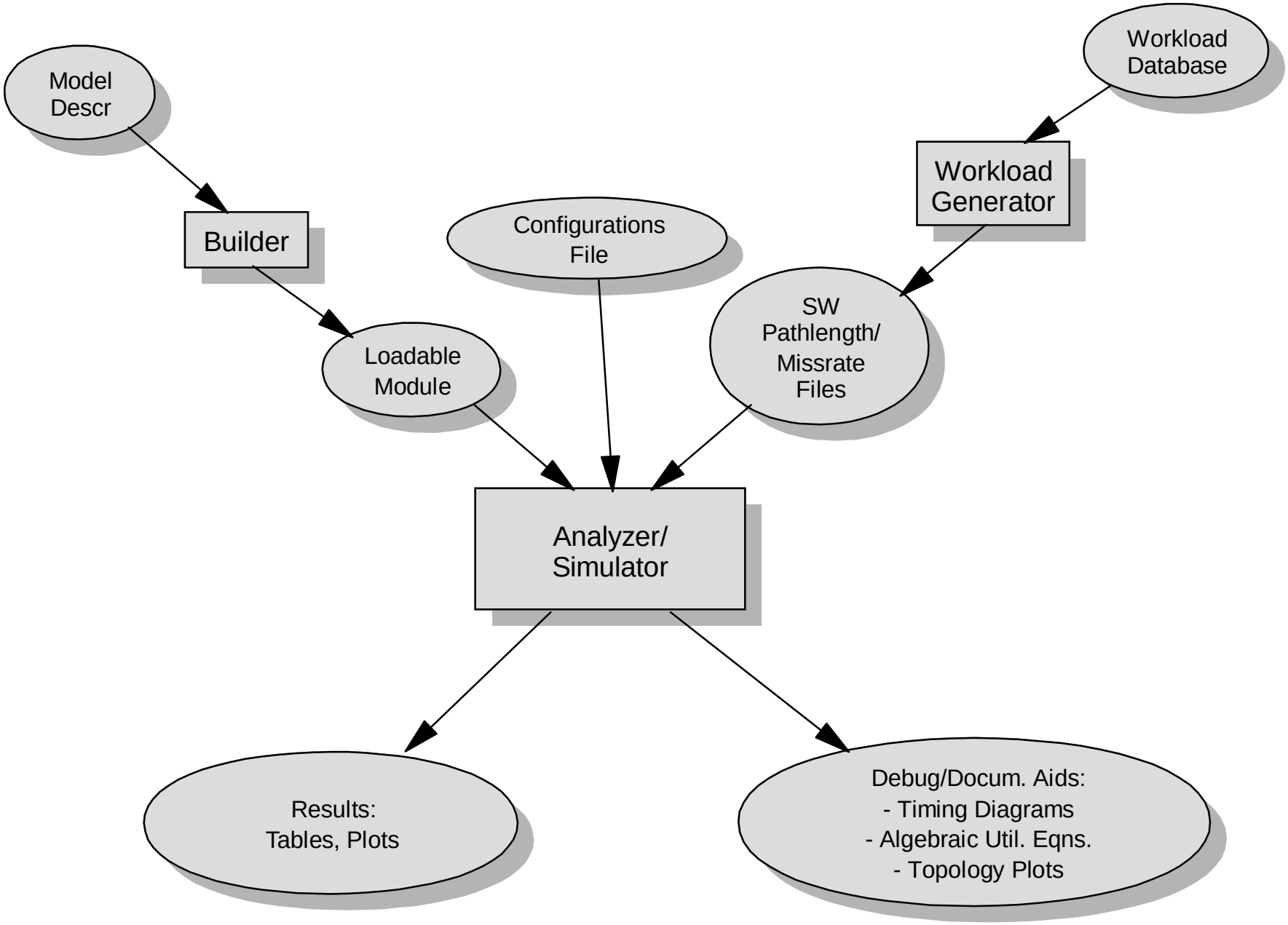
Approach: Queueing Models

- easy to write and modify
- fast (analytic solution)
- good for gross tradeoffs of cache structures, bus bandwidths, topologies
- can be accurate: 5 to 10% (with good workload data!)
- can support a wide range of modeling detail: can approach that of simulation models
- validation aid for simulation models
- fosters a "crisp" understanding of what is important in a system model

Sysmodel: A Solver/Analyzer for Queueing Network Models

- no suitable vendor offerings for analytic modeling
- manually writing analytic model equations is cumbersome and error-prone: it is much better to have a tool to take a high level description and create the equations automatically
- with a carefully designed high level description language, it is possible to automatically generate a simulation model from the same description that is used for the analytic model
- a good infrastructure around an analyzer/simulator adds a great deal of leverage:
  - automatic miss rate loading from workload generator
  - concise specification of configurations to analyze
  - automatic timing diagram generation
  - sophisticated table and chart generation

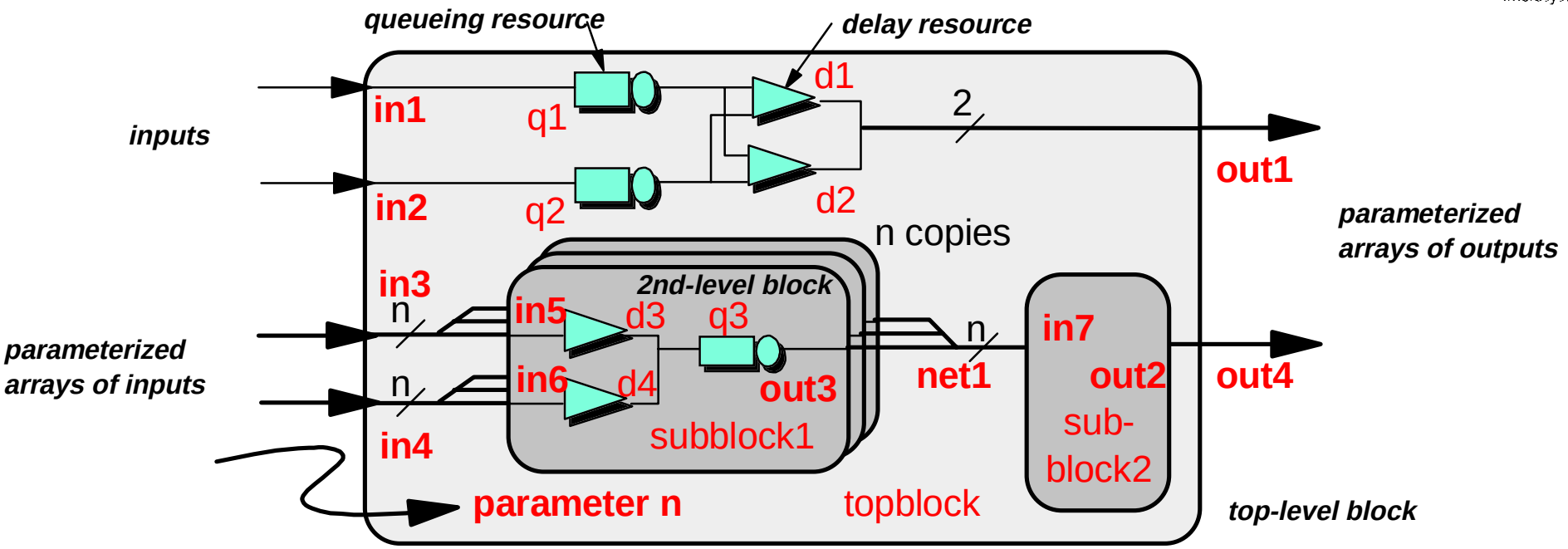
Sysmodel Modeling Environment



Sysmodel Characteristics

- natural representation of memory hierarchy models: textual description of timing diagrams (cache coherence transactions)
- built on top of (embedded in) an existing programming language (ie. C or java):
  - exploit existing compiler, debugger, IDE, libraries, etc.
  - less work than creating an entirely new language
- modularity:
  - support for hierarchically interconnected blocks
  - it should not be necessary to recompile all components for a minor change to only one component
- "first-class" parameter support:
  - all parameters registered in a simulator database
  - enforces better model structure
  - better documentation
  - used in tables of results

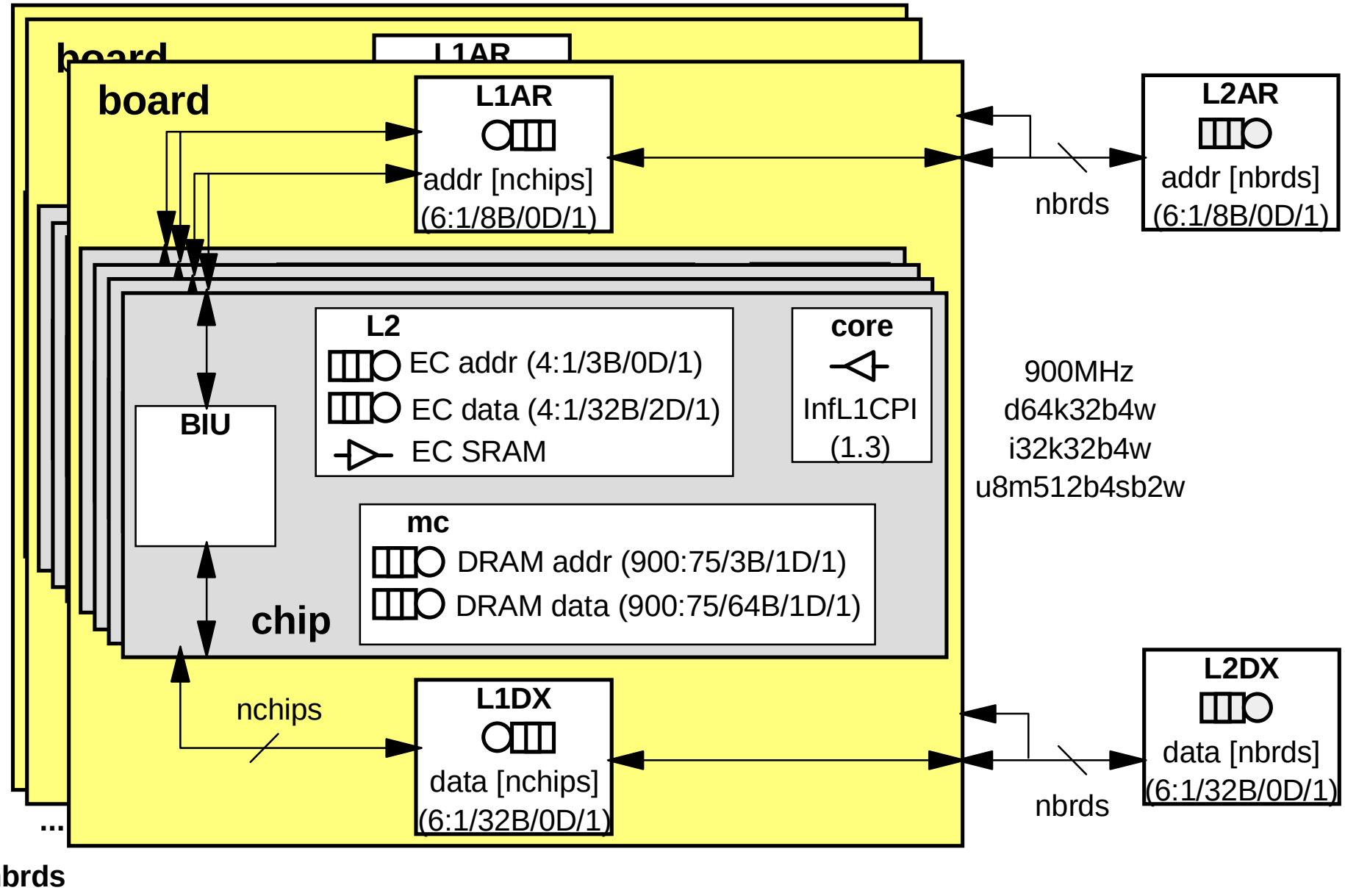
Structural Primitives



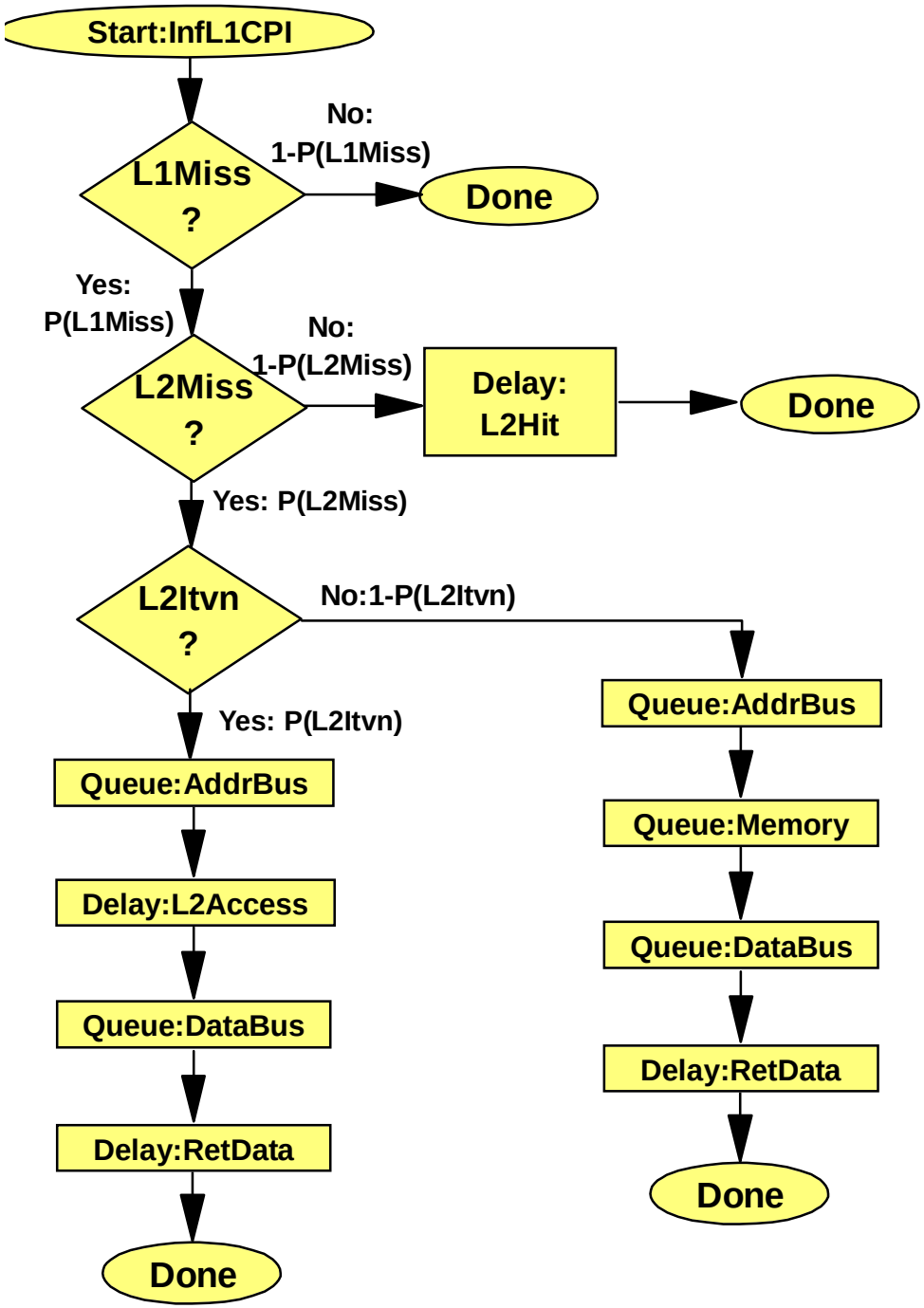

```
BlockDef(subblock1) {
    In(in5, in5_type);
    In(in6, in6_type);
    Out(out3, out3_type);
    Resource(q3, 1);
    Resource(d3, 1e9);
    Resource(d4, 1e9);
}

BlockDef(subblock2) {
    ...
}
```

```
DefBlock(topblock, STRUCTPARM n) {
    int i;
    In(in1, in1_type);
    Out1D(out1, out1_type, 2);
    Resource(q1, 1);
    Net1D(net1, out3_type, n);
    ...
    InstBlock1D(subblock1, subblock1, n);
    InstBlock(subblock2, subblock2);
    for(i=0; i<n; i++) {
        ConnectInput(net1[i], subblock2, in7[i]);
        ConnectOutput(subblock1[i], out3, net1[i]);
        ...
    }
    ...
```

Serengeti Model Structure

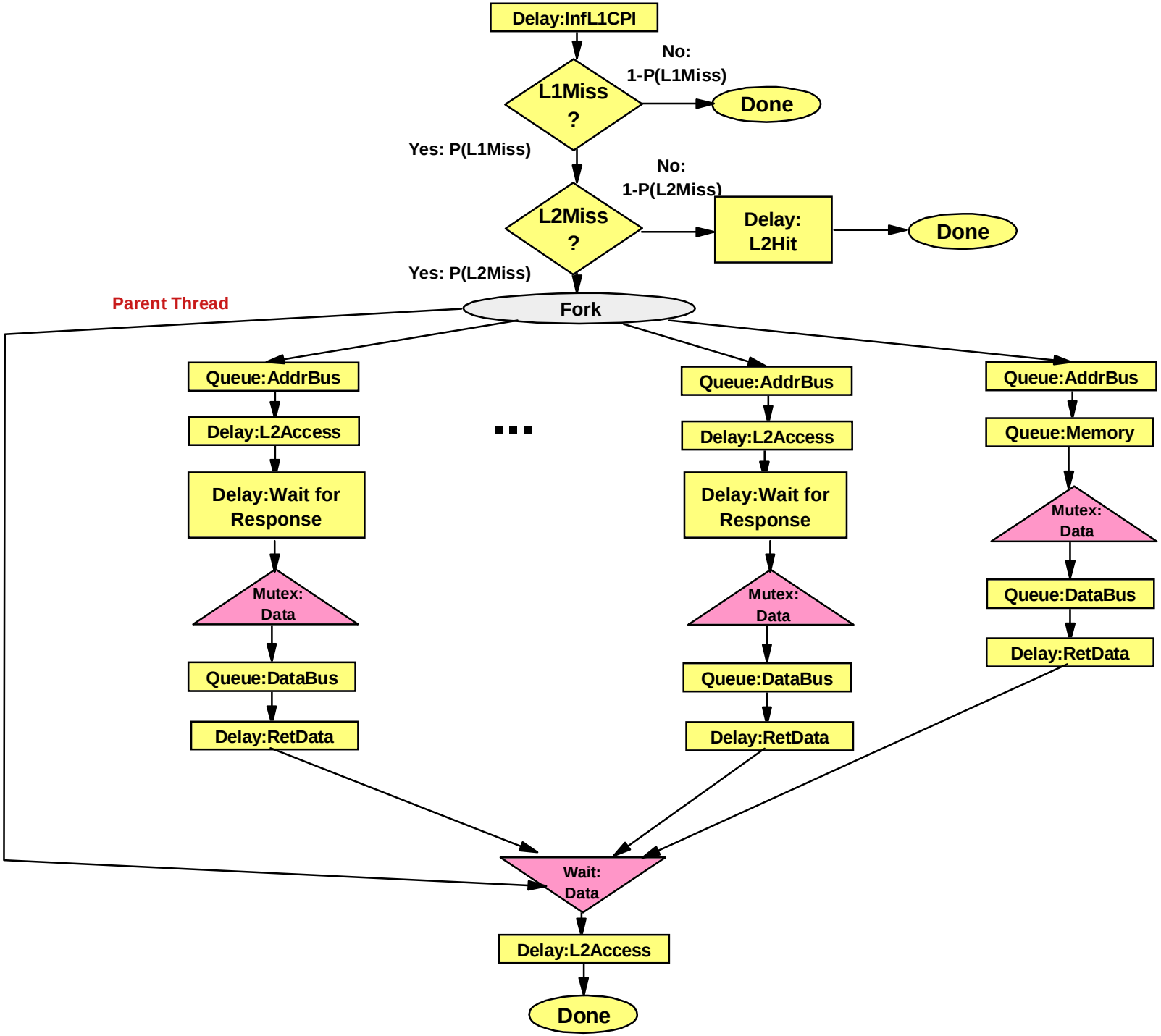


Queueing Network Routing




```
Routing() {
    Start(CPU, InfL1CPI);
    Switch(); {
        Case(1-P_L1Miss); {
            Done();
        } EndCase();
        Default(); {
            Switch(); {
                Case(1-P_L2Miss); {
                    Delay(T_L2Hit);
                } EndCase();
                Default(); {
                    Switch(); {
                        Case(P_L2Itvn); {
                            /* L2 Intervention */
                            Queue(AddrBus);
                            Queue(Memory);
                            Queue(DataBus);
                            Delay(RetData);
                        } EndCase();
                        Default(); {
                            /* go to memory */
                            Queue(AddrBus);
                            Queue(Memory);
                            Queue(DataBus);
                            Delay(RetData);
                        } EndDefault();
                    } EndSwitch();
                } EndDefault();
            } EndSwitch();
        } EndDefault();
    } EndSwitch();
}
```

Sketch of Serengeti Routing



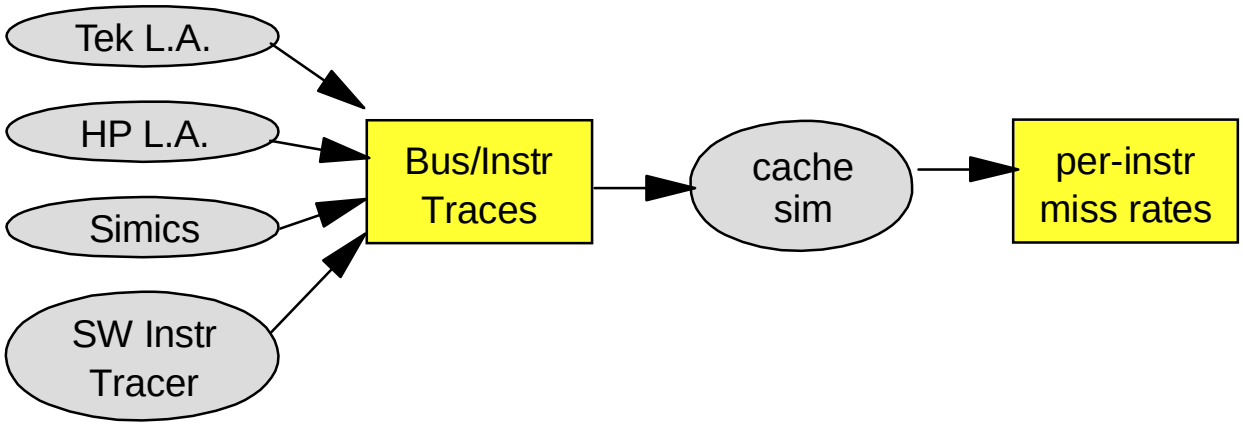
MVA Solver

- compute rates and classes from transaction flow graph:
  - assign visit rates to each node in graph
  - identify customer classes at each queueing center
  - find subgraphs for each Send/Receive and Mutex/WaitMutex construct
- apply "classical" mean value analysis for product-form queueing networks, plus these approximations:
  - Schweitzer's approximation to break recursion
  - Reiser's approximation for deterministic service
  - Heidelberger & Trivedi approximation for asynchronous background traffic
  - a simple heuristic for Fork/Join (Send/Receive) behavior

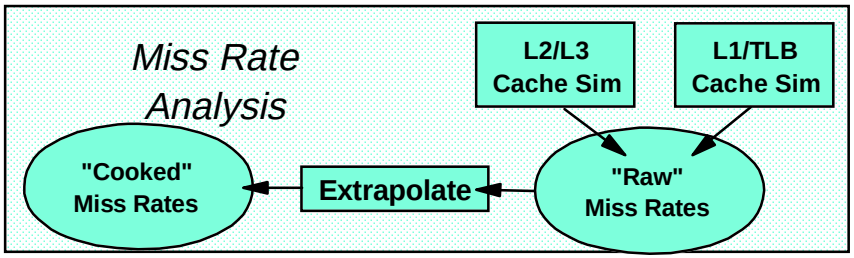
Workload Characterization:

- Miss Rate Analysis

How Hard Can This Be?

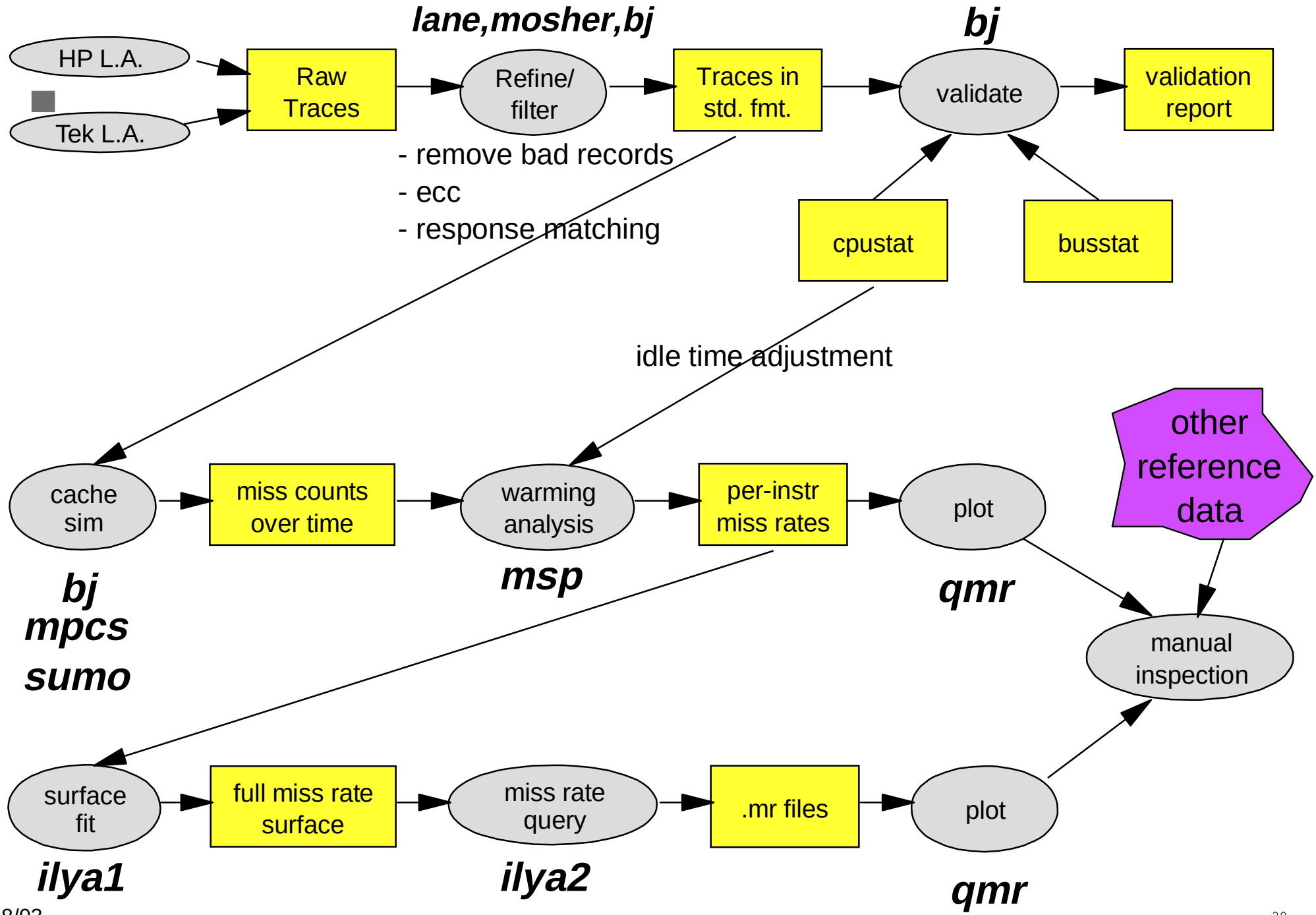


Workload Characterization: Miss Rate Analysis



- estimation of miss rates for cache configurations that we cannot directly simulate (ie. more processors than available in traces)
- includes additional manipulation of cache simulator output to:
  - account for miss rate increases due to increased multiprogramming level ("MPL-effect")
  - quickly estimate miss rates for cache configurations that haven't been simulated (but could be if we had the time)
- multidimensional curve fitting with dimensions of:
  - cache size
  - line size
  - associativity
  - sector size
  - sharing
  - # threads
- automatic fitting tools with seamless interface to sysmodel

How Hard Can This Be?



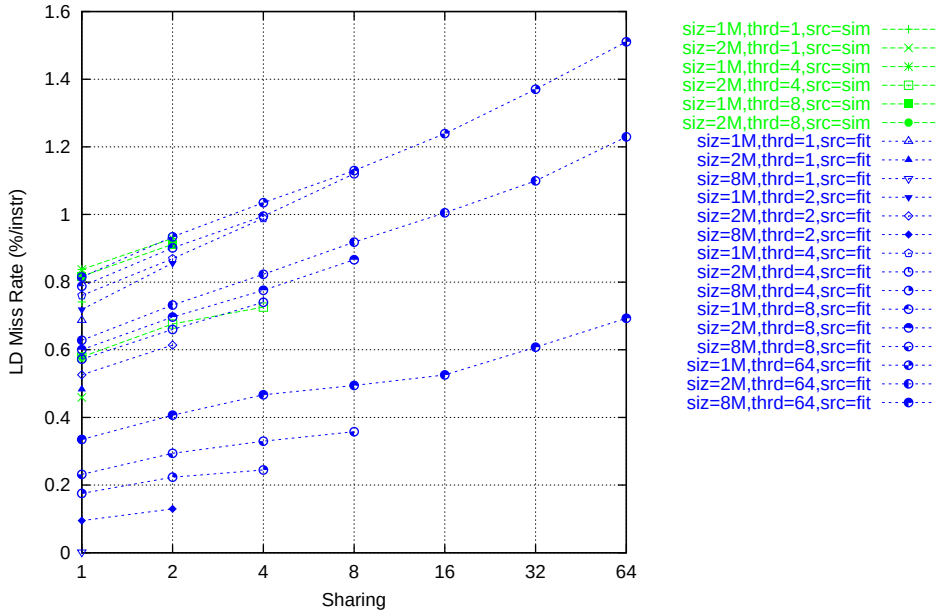
Miss Rate Surface Fitting

- Hastie and Tibshirani "Generalized Additive Models": *Curve Fitting on Steroids*
- 10 miss rate components to model (clean, cache-to-cache, writeback, each per load/store/i-fetch)
- separate multivariate model per rate
- get sparse set of (mr_i, $\underline{c}_i$): mr_i is rate observed for cache configuration $\underline{c}_i$ = *(size, line, subline, nthreads, sharing, assoc)*
- fit an additive model with low order interactions:

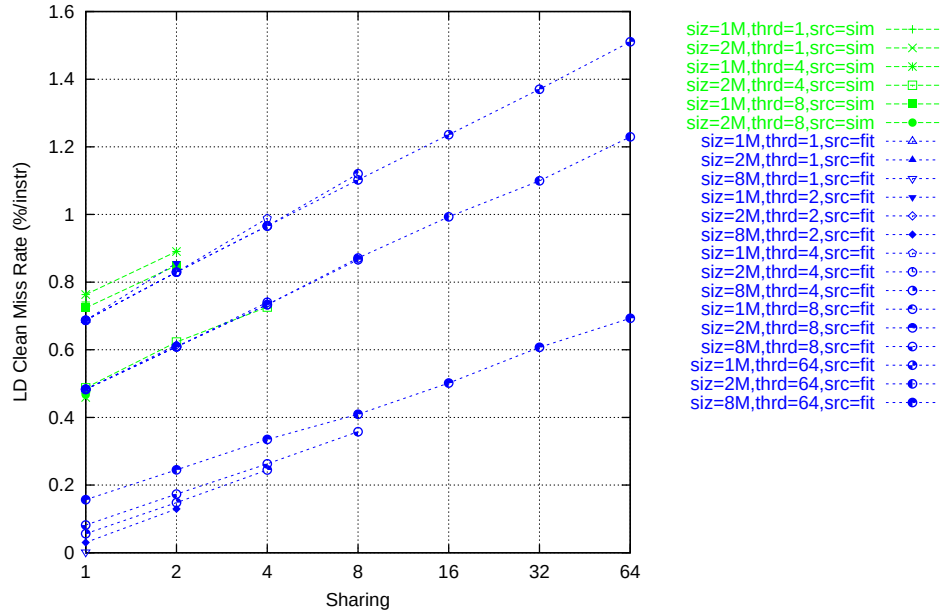
$$f(\underline{x}) = \Sigma_{p=1,P}\, f_p(x_p) + \Sigma_{p<r}\, f_{pr}(x_p, x_r)$$

- extensions for ensuring monotonicity
  ie: clean miss rates never get worse as cache size increases

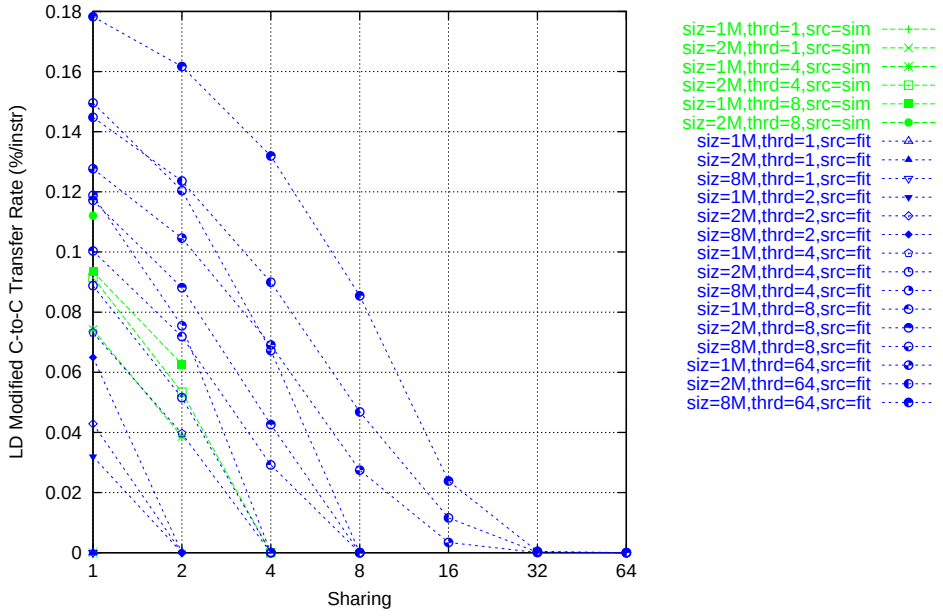
LD Miss Rate vs. Sharing
(Associativity=2, LineSize=64, SubLine=64, Protocol=mosi, Detail=lru_wt)



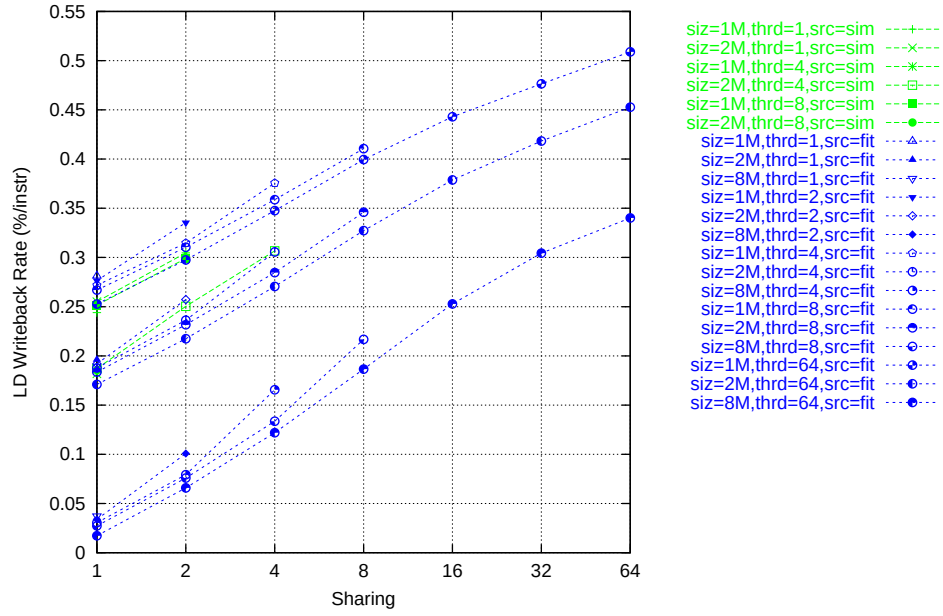
LD Clean Miss Rate vs. Sharing
(Associativity=2, LineSize=64, SubLine=64, Protocol=mosi, Detail=lru_wt)



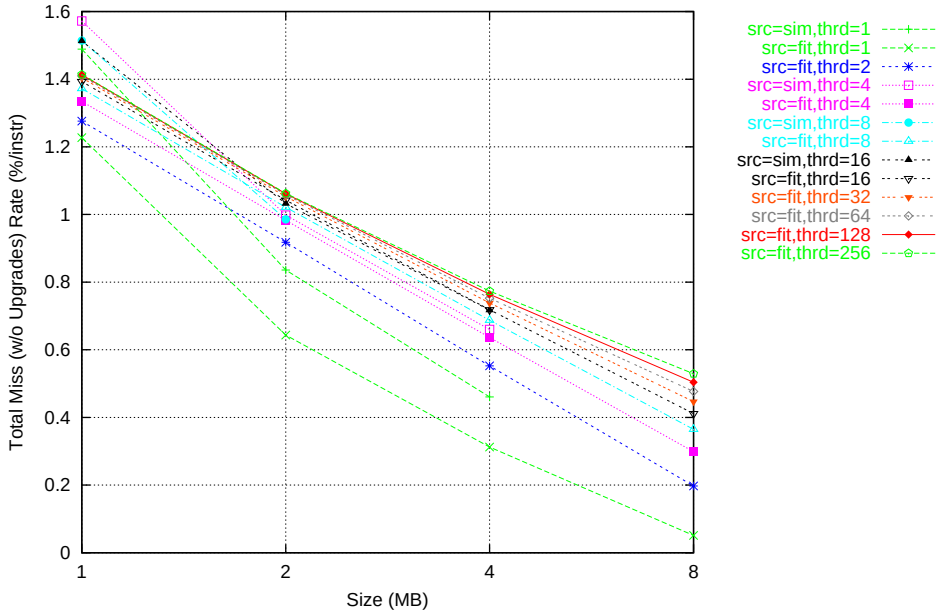
LD Modified C-to-C Transfer Rate vs. Sharing
(Associativity=2, LineSize=64, SubLine=64, Protocol=mosi, Detail=lru_wt)



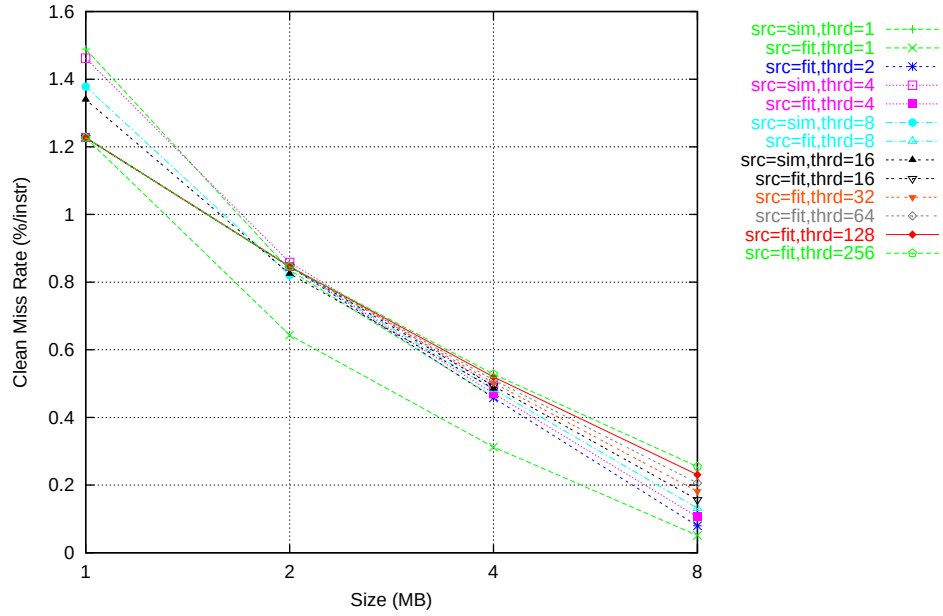
LD Writeback Rate vs. Sharing
(Associativity=2, LineSize=64, SubLine=64, Protocol=mosi, Detail=lru_wt)



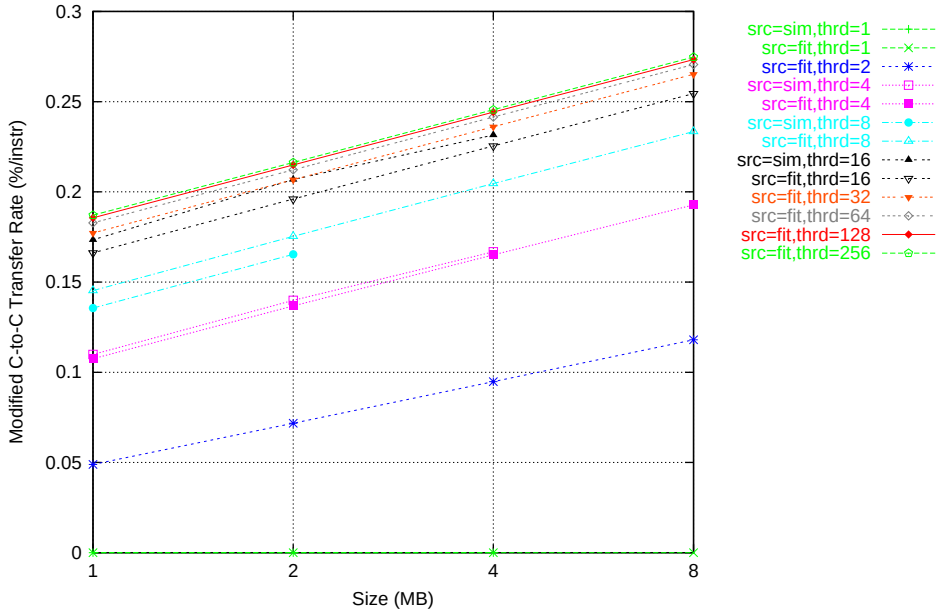
Total Miss (w/o Upgrades) Rate vs. Size (MB)
(Associativity=2, LineSize=64, SubLine=64, Sharing=1, Protocol=mosi, Detail=lru_wt)



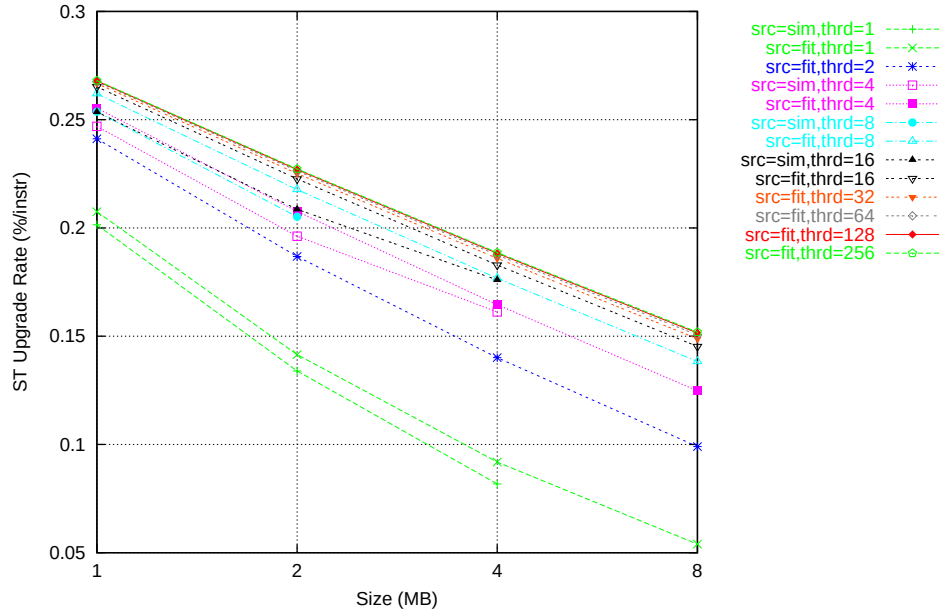
Clean Miss Rate vs. Size (MB)
(Associativity=2, LineSize=64, SubLine=64, Sharing=1, Protocol=mosi, Detail=lru_wt)

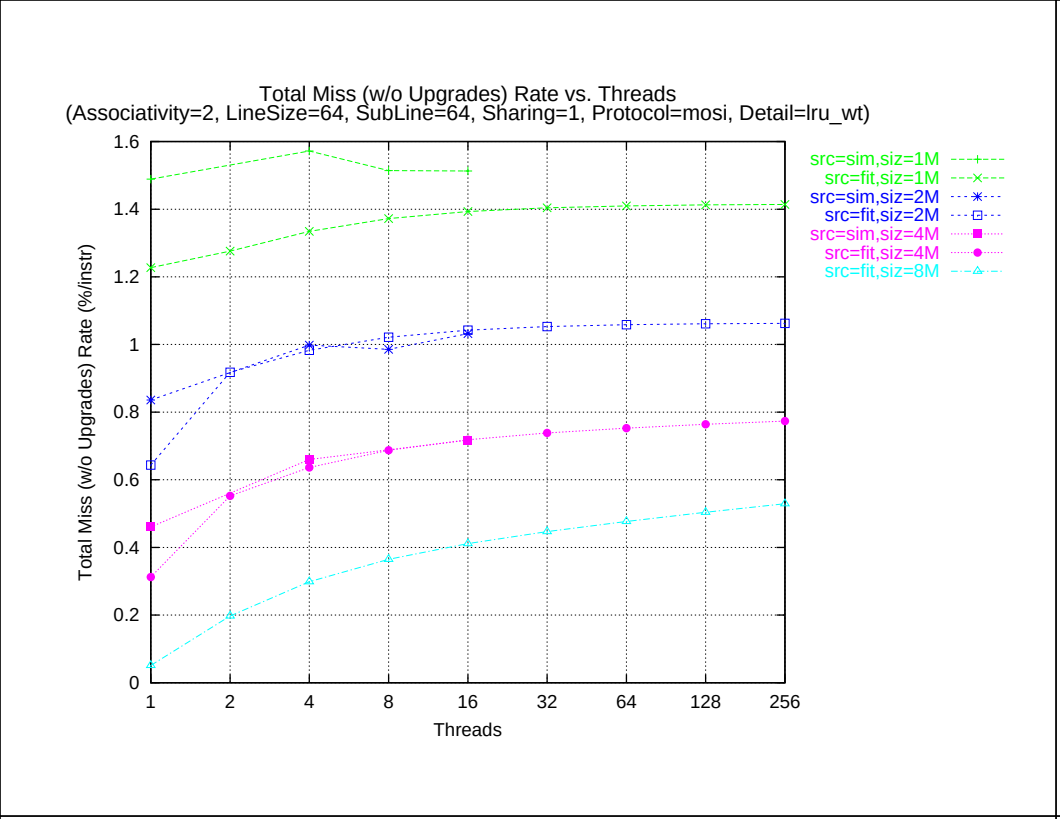


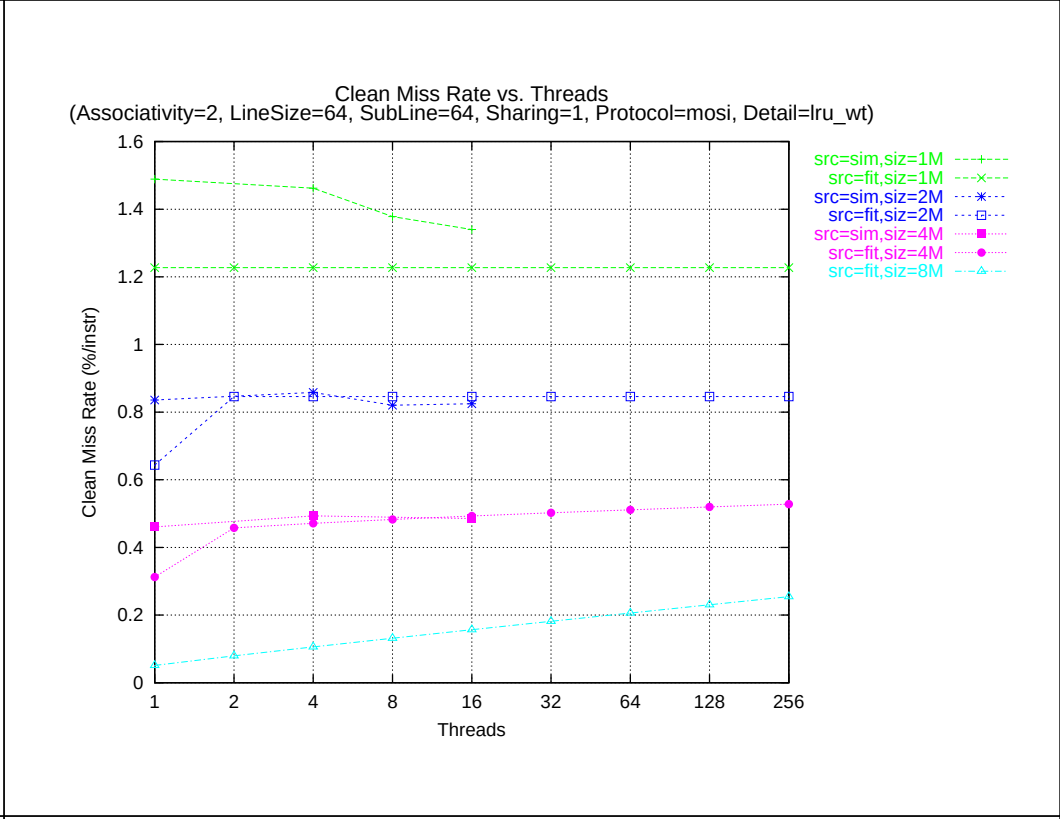
Modified C-to-C Transfer Rate vs. Size (MB)
(Associativity=2, LineSize=64, SubLine=64, Sharing=1, Protocol=mosi, Detail=lru_wt)

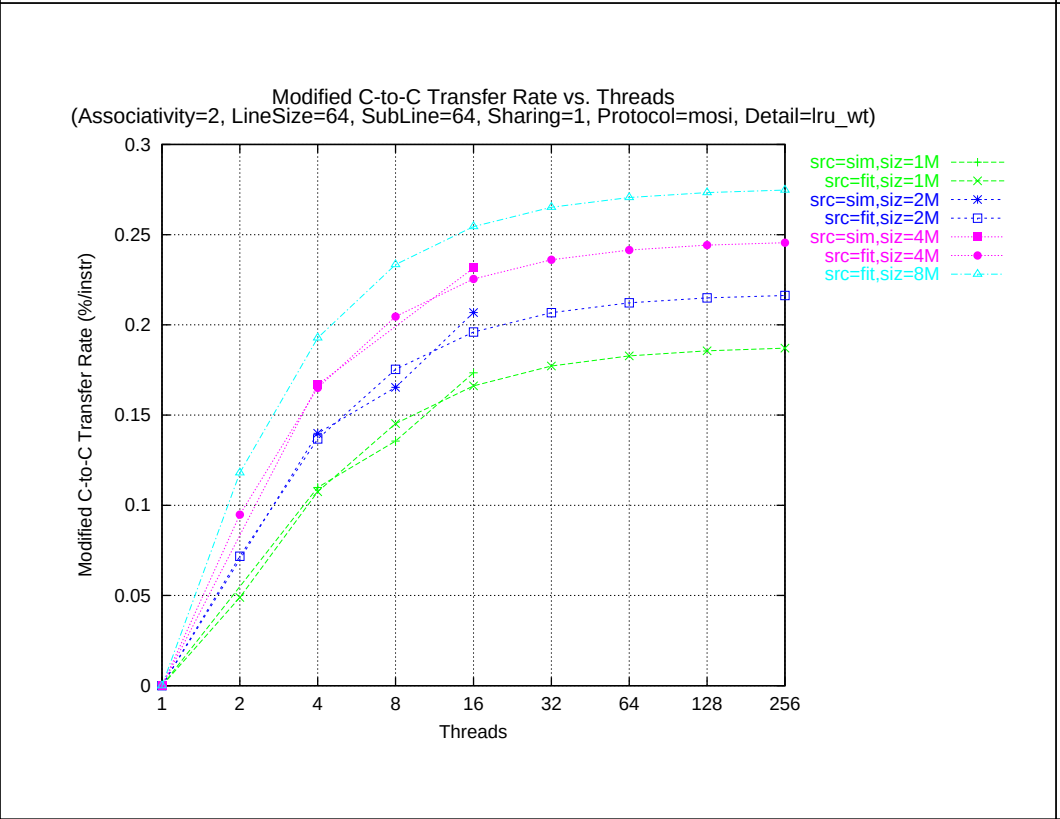


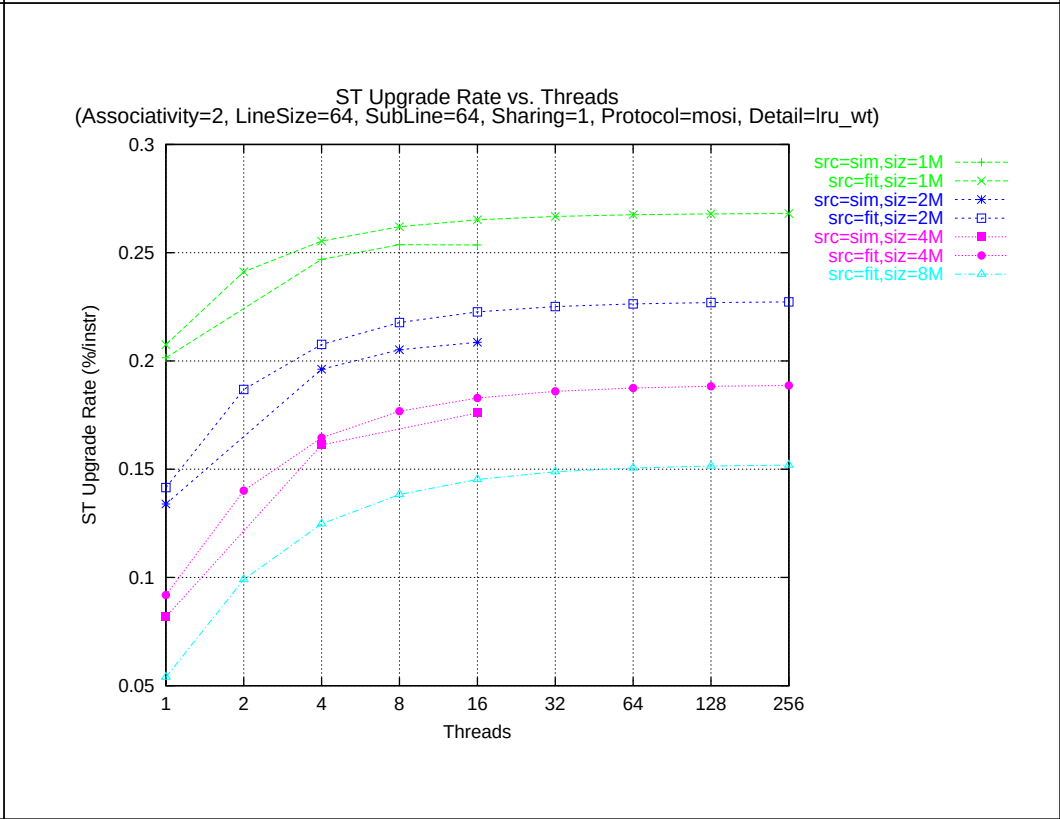
ST Upgrade Rate vs. Size (MB)
(Associativity=2, LineSize=64, SubLine=64, Sharing=1, Protocol=mosi, Detail=lru_wt)




Total Miss (w/o Upgrades) Rate vs. Threads
(Associativity=2, LineSize=64, SubLine=64, Sharing=1, Protocol=mosi, Detail=lru_wt)
Total Miss (w/o Upgrades) Rate (%/instr)
Threads
src=sim,siz=1M
src=fit,siz=1M
src=sim,siz=2M
src=fit,siz=2M
src=sim,siz=4M
src=fit,siz=4M
src=fit,siz=8M


Clean Miss Rate vs. Threads
(Associativity=2, LineSize=64, SubLine=64, Sharing=1, Protocol=mosi, Detail=lru_wt)
Clean Miss Rate (%/instr)
Threads
src=sim,siz=1M
src=fit,siz=1M
src=sim,siz=2M
src=fit,siz=2M
src=sim,siz=4M
src=fit,siz=4M
src=fit,siz=8M


Modified C-to-C Transfer Rate vs. Threads
(Associativity=2, LineSize=64, SubLine=64, Sharing=1, Protocol=mosi, Detail=lru_wt)
Modified C-to-C Transfer Rate (%/instr)
Threads
src=sim,siz=1M
src=fit,siz=1M
src=sim,siz=2M
src=fit,siz=2M
src=sim,siz=4M
src=fit,siz=4M
src=fit,siz=8M

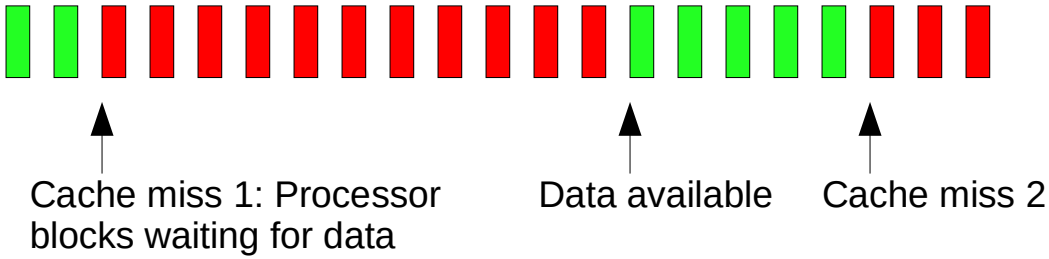

ST Upgrade Rate vs. Threads
(Associativity=2, LineSize=64, SubLine=64, Sharing=1, Protocol=mosi, Detail=lru_wt)
ST Upgrade Rate (%/instr)
Threads
src=sim,siz=1M
src=fit,siz=1M
src=sim,siz=2M
src=fit,siz=2M
src=sim,siz=4M
src=fit,siz=4M
src=fit,siz=8M

Workload Characterization:

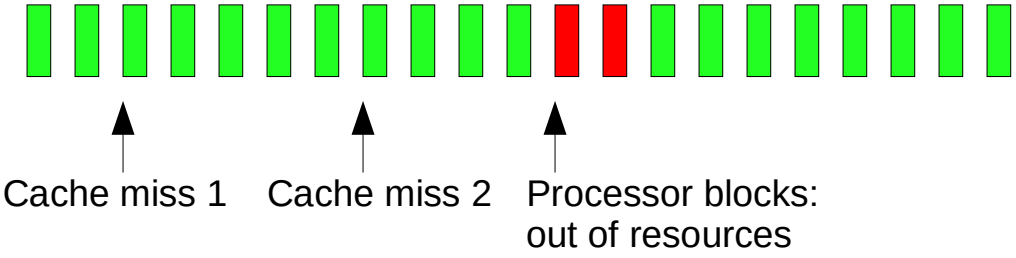
- Processor Abstraction

Overview

- How to model memory level parallelism?
- Processor with no memory level parallelism:



- Processor with memory parallelism:



Model

- Simple processor model (h stands for hit rate, p for penalty, and n for number of cache levels):

$$CPI = CPI_{\infty} + \sum_{i=0}^{n-1} h_i \; p_i$$

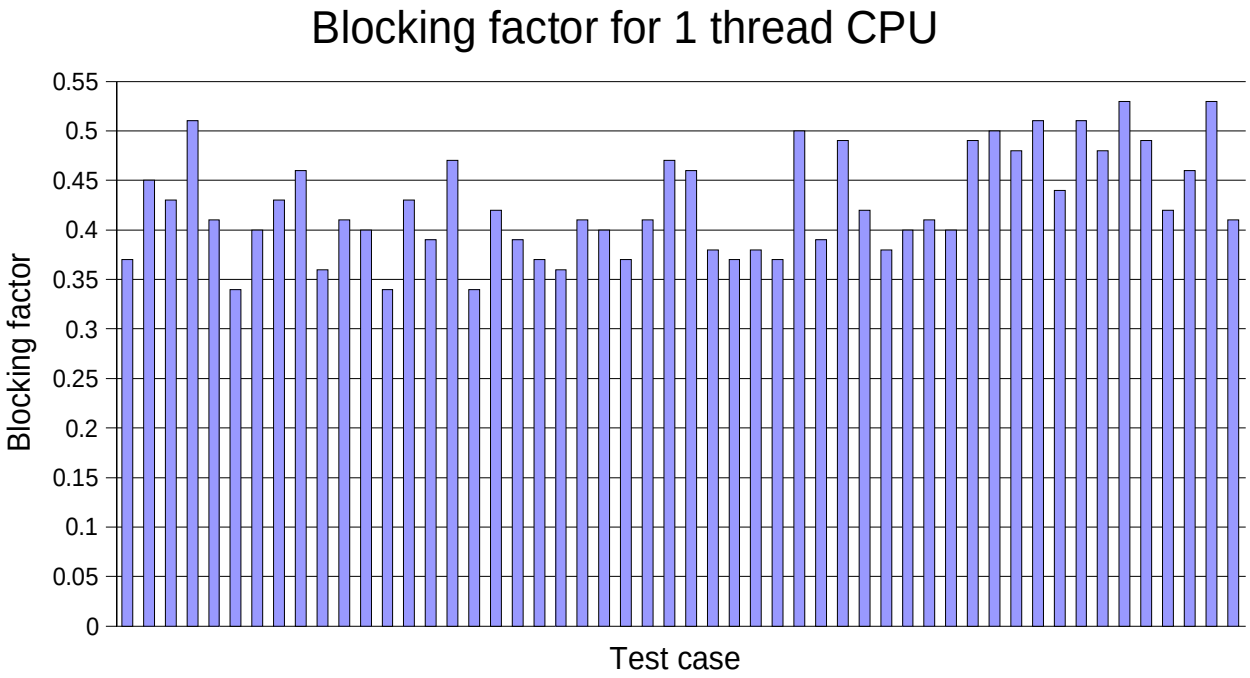
- Model for processor with parallelism (f stands for blocking factor):

$$CPI = CPI_{\infty} + \sum_{i=0}^{n-1} h_i \; f_i \; p_i$$

Questions

- Does blocking factor depend on
  - latency?
  - cache level?
  - access type (data/instruction load/store)?
  - miss rates?

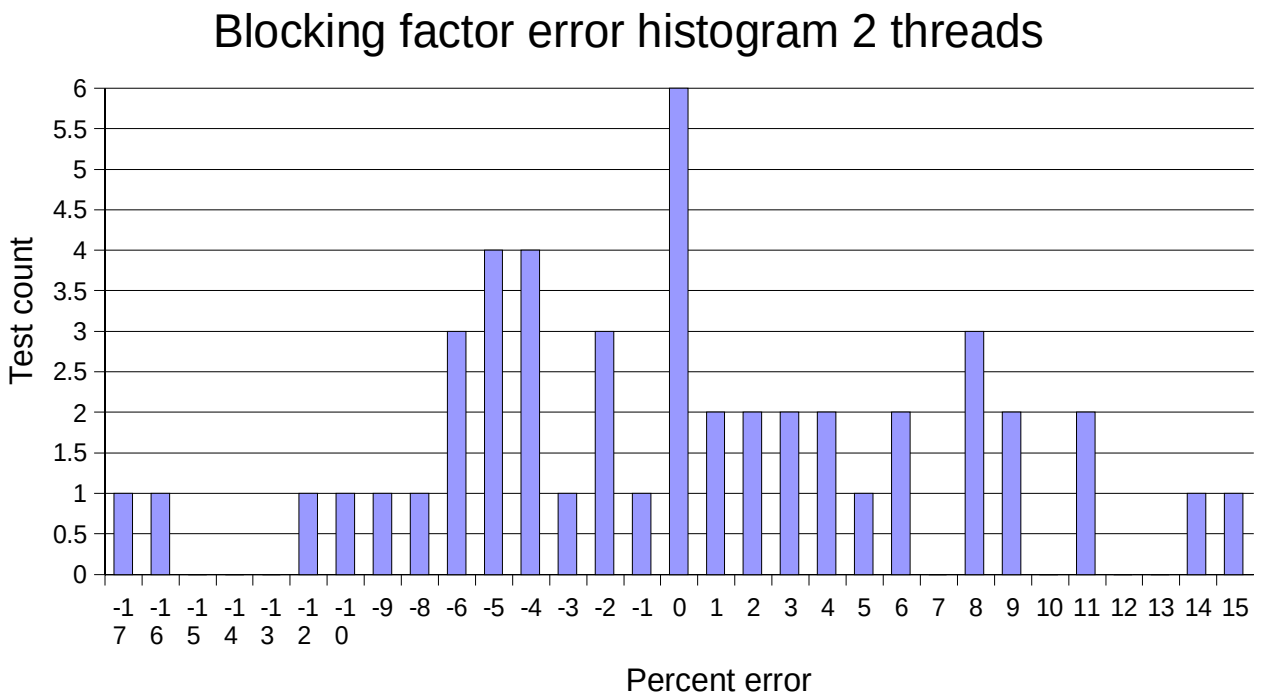
Results


Blocking factor for 1 thread CPU
Blocking factor
0.55
0.5
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
Test case

Results

- Average block factors
  - 1 thread CPU: 0.42
  - 2 thread CPU: 0.45
- Infinite L1 CPI:
  - 1 thread CPU: 1.47 (1.485 measured)
  - 2 thread CPU: 1.93 (1.94 measrured)
- Average error in CPI:
  - 1 thread CPU: 6.27%
  - 2 thread CPU: 6.05%

Results


Blocking factor error histogram 2 threads
Test count
6
5.5
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
-17 -16 -15 -14 -13 -12 -10 -9 -8 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Percent error

Observations

- good news for probabilistic modeling of out-of-order processors:

  blocking factor is relatively insensitive to broad variations in miss penalties and cache sizes

- as latency goes to infinity, blocking factor does NOT go to 1

- additional work shows that making blocking factors a (weak) function of latency reduces errors substantially

- ongoing work to understand this further

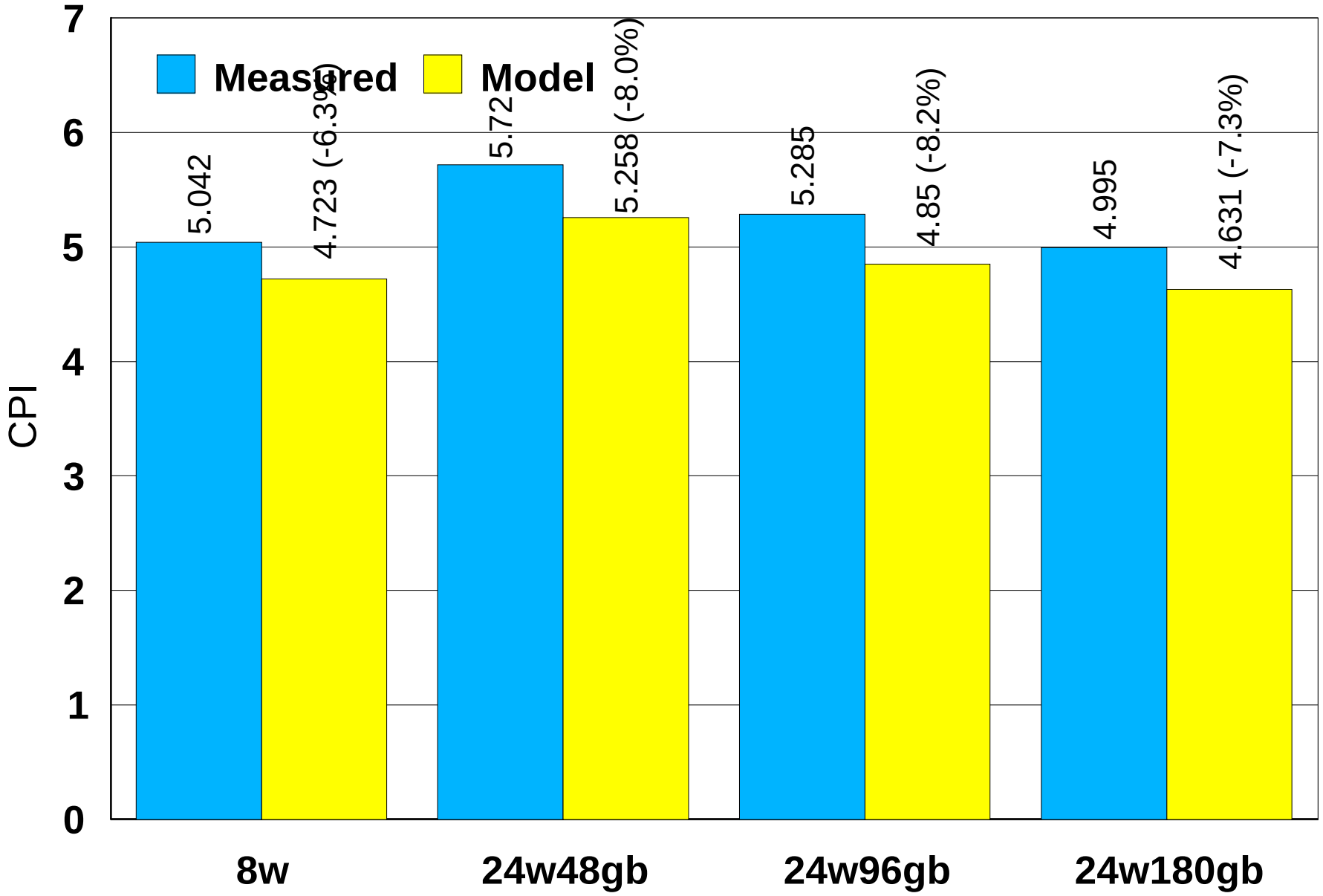
Validation:
Model vs. Measurement

Serengeti Model vs. Measurement

- all configurations:
  - 900MHz Cheetah+ processor, 225MHz L2, 150MHz centerplane
  - u8m64b1w L2
  - Large pages enabled
- 24w Serengeti configurations:
  - Oracle 902, Solaris 5.9 Build 58, log_parallelism = 2
  - data from PAE memory scaling runs
- 8w Serengeti configuration:
  - Oracle 902 process model, s9

Name	# Procs	tpmC (K)	SGA (GB)	CPI	Pathlength (Minstr)	IO's/Trans
8w	8	43.5	14	5.033	0.8821	9.24
24w_48	24	102	48	5.720	0.9847	9.11
24w_96	24	130	96	5.285	0.8368	5.91
24w_180	24	142	180	4.996	0.8113	4.68

TpmC CPI: Model vs. Measured



Observations

- good correlation (within 8%)
- highest utilization is about 50% (system address busses), so system experiences low queueing congestion
- why is model optimistic?
  - are miss rates from counters correct?
  - are static latencies correct?
  - bursty memory accesses?
  - kernel cage effects?
  - ?

Conclusion

Conclusion

- Goal: timely, accurate performance projections for Sun systems
  - changing workloads
  - paucity of traces
  - need for early design space exploration
  - many, many configurations to analyze
- hierarchical models are effective
  - detailed timers for processors
  - queueing network models for system
  - point models for miss rate analysis
- modeling accuracy and breadth of design space determined by workload characterization methodology:
  - bus traces and instruction traces
  - thorough validation
  - automatic surface fitting

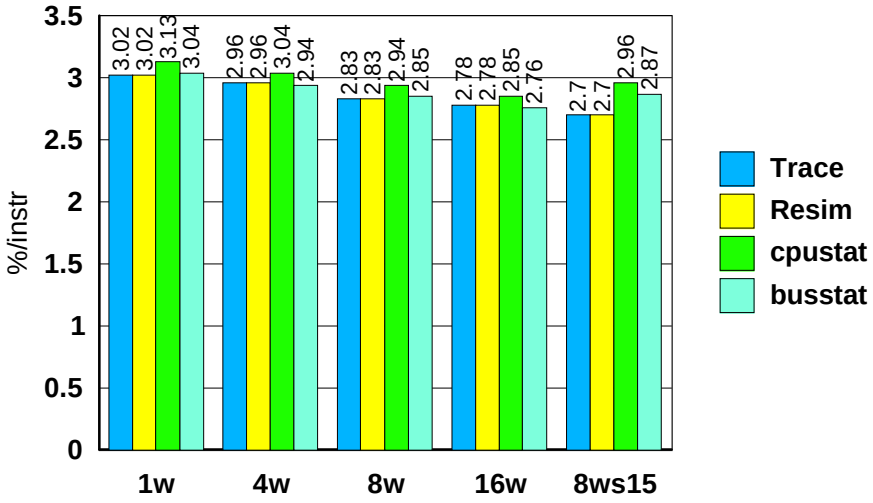
Future Directions

- extend techniques to clustered, multi-tiered systems
- extend techniques to large floating-point computation architectures
- refine processor abstractions

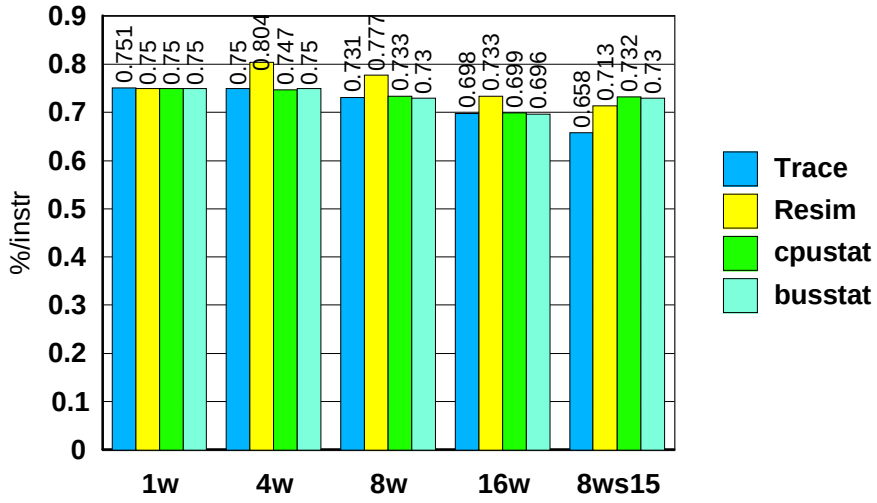
Backup Slides

Tpcc Bus Trace Validation Results
u512k64b1w MOESI

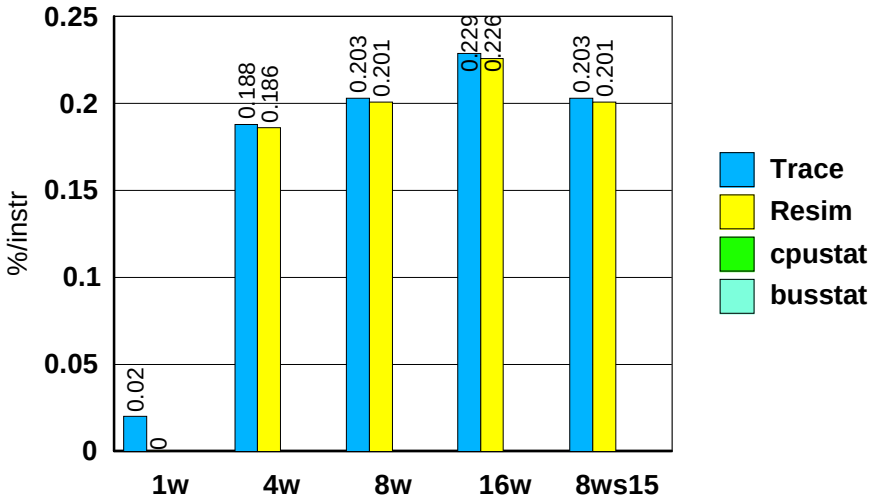
Total Miss Rate



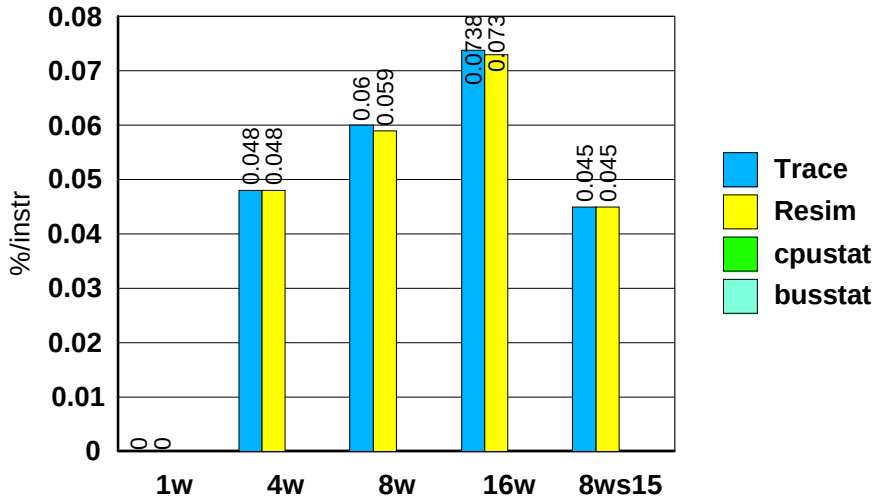
Writeback Rate



Cache-to-Cache Transfer Rate



Upgrade Rate



Sketch of Solver Algorithm

/* prepare graph */

- assign visit rates to each node
- identify customer classes at each queueing center
- find subgraphs for each Send/Receive and Mutex/WaitMutex construct

/* solve the network for response time R (= cpi) */

$R_{new} = 10$; /* a guess */
repeat {
  $R_{old} = R_{new}$
  for (each queue *i) {*
    $R_i = S_i + S_i * Q_i(N-1)$
    $Q_i = N/R_{old} * V_i * R_i$ /* Little's Law: $L = \lambda * W$ */
  }
  - traverse routing graph and compute R_{new}
} until ($|R_{new} - R_{old}| < \delta$)

/* traversing routing graph to compute R_{new}: */

- using depth-first search, compute R_{thread} for each thread as follows:
  - at visit nodes (to queue *i*):
    $R_{thread} \mathrel{+}= V_i * R_i$
  - at Receive(min) nodes:
    $R_{thread} = \max(R_{thread}, \min\{R_{send,j}\})$
  - at Receive(max) nodes:
    $R_{thread} = \max(R_{thread}, \max\{R_{send,j}\})$
  - at WaitMutex nodes:
    $R_{thread} = \max(R_{thread}, \Sigma_j V_{mutex,j} * R_{mutex,j})$
- R_{new} is R_{thread} for root thread

Basic MVA

/* single class */

$R_{new} = 10$; /* a guess */
repeat {
  $R_{old} = R_{new}$
  for (each queue *i) {*

    $R_i = S_i + S_i * Q_i(N-1)$
    $Q_i = N/R_{old} * V_i * R_i$ /* Little's Law: $L = \lambda * W$ */

  }
  $R_{new} = \Sigma_i \ R_i$
} until ($|R_{new} - R_{old}| < \delta$)

$Q_i(N-1)$ here means Q_i in same network, but with 1 fewer customer.

/* Schweitzer's approx. to break recursion */

Assume $Q_i(N-1)/(N-1) \approx Q_i(N)/N$

Hence: $Q_i(N-1) \approx (N-1)/N * Q_i(N) \equiv (N-1)/N * Q_i$

Ref: P. Schweitzer, "Approximate Analysis of Multiclass Closed Networks of Queues", *JACM*, 1981.

/* multi-class */

$R_{new} = 10$; /* a guess */
repeat {
  $R_{old} = R_{new}$
  for (each queue *i) {*
    for (each class *c*) {

      $R_{i,c} = S_{i,c} + \Sigma_{k \neq c} S_{i,k} * Q_{i,k}(\mathbf{N-1_c})$
      $Q_{i,c} = N_c/R_{old} * V_{i,c} * R_{i,c}$ /* $L = \lambda * W$ */

    }
  }
  $R_{new} = \Sigma_i \ \Sigma_c \ R_{i,c}$
} until ($|R_{new} - R_{old}| < \delta$)

/* multiclass approx. to break recursion */

$$Q_{i,k}(\mathbf{N-1_c}) \approx (N_c-1)/N_c * Q_{i,c}(\mathbf{N}) + \Sigma_{k \neq c} Q_{i,k}(\mathbf{N})$$

Additional MVA Approximations

/* Deterministic Service */

instead of:

$$R_i = S_i + (N-1)/N * S_i*Q_i$$

use:

$$R_i = S_i + (N-1)/N * [(Q_i - B_i)*S_i + B_i * S_i/2]$$

where B_i is the probability that an arriving customer finds the server busy, approximated by:

$$B_i = (U_i - U_i/N)/(1 - U_i/N)$$

Ref: M. Reiser, "A Queueing Network Analysis of Computer Communication Networks with Window Flow Control", *IEEE Trans. Comm.,* 1979.

/* Asynchronous Background Traffic */

- same as non-background traffic, except do not include $R_{i,c}$ for background classes in summation for R_{new}

Ref: P. Heidelberger and K. S. Trivedi, "Queueing Network Models for Parallel Processing with Asynchronous Tasks", *IEEE Trans. on Computers*, 1982.

Additional MVA Approximations

/* Send/Receive (Fork/Join) */

Assume E(min(A,B)) = min(E(A), E(B))
or E(max(A,B)) = max(E(A), E(B))

- at Receive(min) nodes:
  $R_{thread} = \max(R_{thread}, \min\{R_{send,j}\})$

- at Receive(max) nodes:
  $R_{thread} = \max(R_{thread}, \max\{R_{send,j}\})$

- Why do we expect this to be a reasonable approximation?
  - in multiprocessor models, service distributions are usually deterministic, and utilizations are often not extreme
- The starcat model (TBD) will be the first real test of this approximation.
- Simulation feature of sysmodel will be used for validation.