INTEGRATION AND PERFORMANCE EVALUVATION OF SYNSOPSYS MEMORY MODULE (MEM) TO CUSTOMSIM

Under the guidance of:

Dr.Amelia Shen

Director R&D

Design Group

SYNOPSYS Inc

By:

Sathyanarayanan Ramesh Babu

Technical intern

Design Group

SYNOPSYS Inc

PREFACE

This document is a part of my Summer Internship Program at SYNOPSYS Inc. The document makes an account of our efforts and the things that I have learned over the period of this internship program, from JUNE-8-2015 to AUG-21-2015. The intent behind drafting this document is two fold:

(i) It should serve as a starting point for someone who starts working on interfacing MEM with Customsim. Ideally, if that person is of a limited exposure to SYNOPSYS and EDA, like I was when I started in June, this document will be of some value.

(ii) While articulating the things I had learned over the period of 3 months at SYNOPSYS, my understanding of the things discussed below gets consolidated.

ACKNOWLEDGEMENTS

I sincerely thank Dr.Amelia Shen (Dir. R&D) for giving me the opportunity to be a part of this wonderful organization and team. My heartfelt gratitude to Dr. Mayukh Bhattacharya (R&D Engg) for giving me his valued inputs on a wide verity of topics. I extend my gratitude to Mrs.Huiping Huang (R&D Engg) for her guidance all through the program. I would also extend my gratitude to Dr.Bharat Joshi (Associate Professor & Graduate Coordinator UNCC) for his continued guidance and support.

Contents:

Intern Final Documentation:

Section1: Organizational structure and Product outline :

The given below tree represents the organizational structure of SYNOPSYS Inc.

We belong to the Design Group which includes the company's implementation and analog/mixed signal product lines.

Customsim is a fast-SPICE mixed signal circuit simulator. We shall discuss more about FAST-spice technology and how it is different from conventional SPICE technology, in the following section.

Section 2: A Perspective of FAST-spice simulation Technology:

In order to appreciate various schemes implemented by Fast-SPICE, we need a basic understanding of any basic SPICE simulation tool. I will attempt to cover some top level ideas that have been implemented in a conventional SPICE simulation tool. Refer [2] from 'sources' for more a detailed background on conventional SPICE tools.
In order to simulate a circuit and solve it to bring about its various parameters the application has to first derive the system equations from a given circuit. These system equations have to be solved. Solving equations from a software perspective is to populate a matrix and iterate it over a solving algorithm to bring about a convergence. There are few details to be noted on the aforementioned SPICE-simulators which are as follows.

A short digression on SPICE simulation:

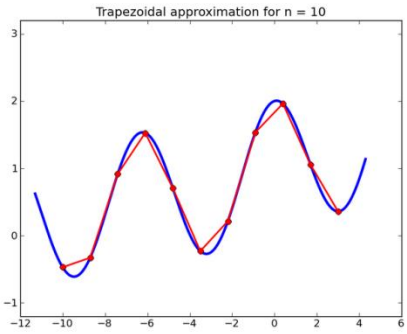
The matrix:

The system equations in SPICE are represented in the form of matrices. The equations characterize the linear representation between the voltage and current for every element in the circuit. The two main types of matrices we are talking about is,

Voltage array and the current array, Voltage array holds the node voltage and current array holds the branch current of the circuit. The voltage array is derived by SPICE using Kirchhoff's law. The conductance array is populated by deriving the linear counterparts of non-linear components of the circuit.

The non-linear components of the circuit is subjected to a process of linearization.[3] As the matrices hold linear parameters, linearization function provides a rational approximation that can be filled into the appropriate conductance matrix.

An example of a piecewise linear approximation is as follows:



$$I_d = I_s\left[\exp\left(\frac{qV_d}{NKT}\right)-1\right]$$

$$I_d = G_d{*}V_d + I_{eq}$$

The following equations represent the linearized diode equation.

These matrices are constructed through inspection. Matrix construction by inspection builds the system matrices and identifies the location of each element in the matrices as soon as the nodes connected to the elements are defined. Matrix construction by inspection builds the system matrix with the help of predefined element templates. Every element in the SPICE has some template associated with it. The

SPICE simulator has a template for a basic resistor and as it comes across more components as a part of the input circuit, it appends the current template and makes it bigger. Likewise, the matrix keeps getting bigger as more and more components are a part of the design.

Example of a template for a basic resistor is as follows.

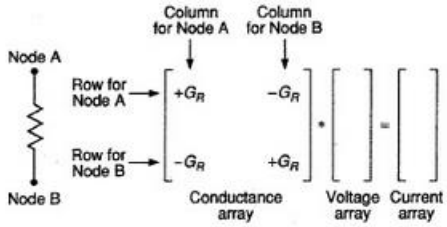


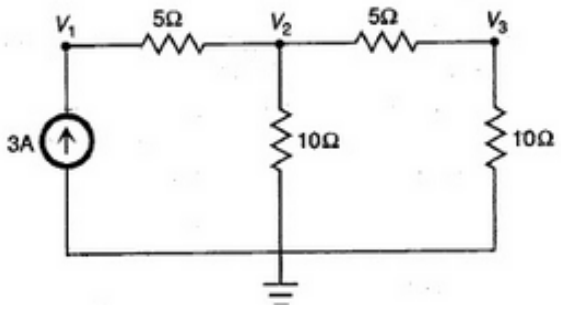
Figure 2.15(a) The template of a resistor.

Example of netlist is as follows:

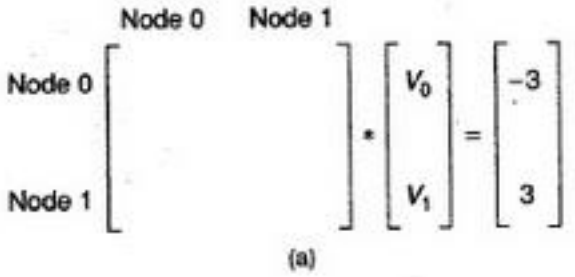
Example Circuit

```
I1 0 1 3A
R1 1 2 5_OHMS
R2 2 0 10_OHMS
R3 2 3 5_OHMS
R4 3 0 10_OHMS
.END
```

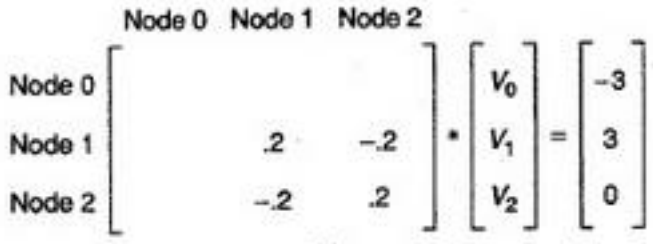
Its corresponding circuit is:



This circuit is incrementally built by the following course of steps:

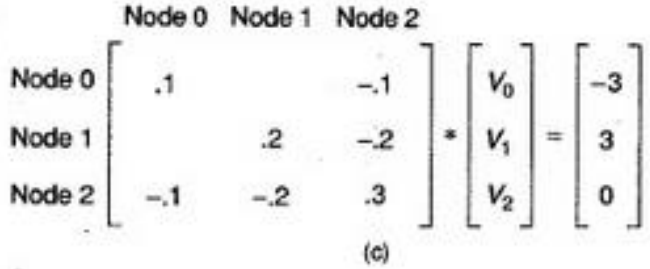


(a)

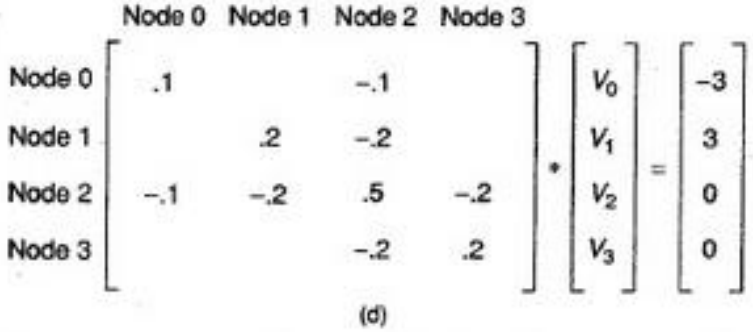


(b)

Figure 2.18(*a*) and (*b*) An example of matrix construction by inspection.



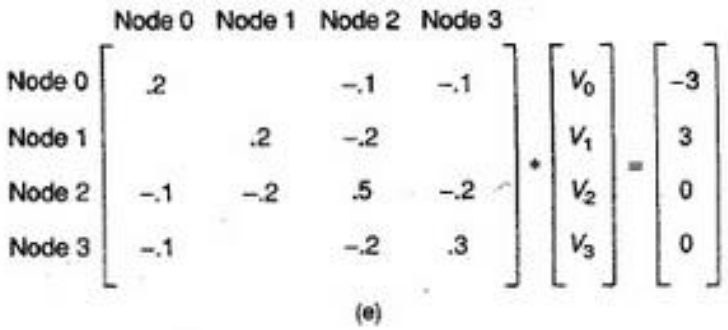
(c)



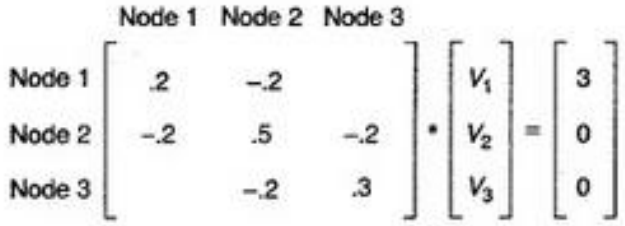
(d)

Figure 2.18(*c*) and (*d*) An example of matrix construction by inspection.

The 4 voltage values are reflective of the 4 nodes in the system. As the input is being parsed, we keep updating the matrices accordingly.



(e)



(f)

Figure 2.18(e) and (f) An example of matrix construction by inspection.

Solving the populated matrices: - If the matrices contain only linear components the solving technique is Gauss-elimination. If there are one or more non-linear components in the circuit, the resulting equations become non-linear (quadratic and so on) the algorithm used to solve such matrices are Newton-Raphson.

While solving the populated matrix in an iterative manner, the solution engine starts iterating with an initial guess for each of the node voltages. For each successive iteration, a new set of node voltages are predicted. The engine monitors the results of the current iteration and previous iteration. Generally due to round off errors, we cannot have ideal convergence. There is a maximum cap for error that is tolerable. There is also a maximum number of iteration permitted. Both these parameters can be controlled by the user. If the error is not less than the maximum error that can be tolerated in the specified number of iterations, the simulation is set have failed.

With the solutions of the equations SPICE does 4 main types of analysis.

(i) DC operating point analysis
(ii) DC sweep analysis
(iii) AC frequency sweep analysis
(iv) Transient time sweep analysis

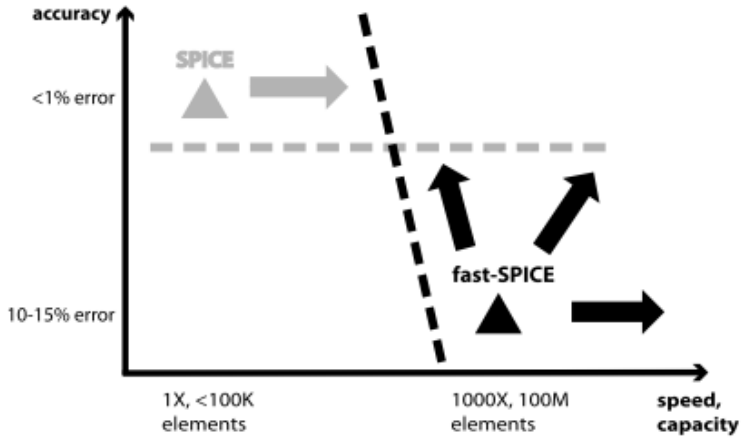
With these basic understanding of SPICE simulators, We now move on to what Fast-SPICE technology aims at doing.

Limitations of SPICE: The main philosophy of SPICE is the amount of accuracy it tries to achieve. Some restrictions posed on a SPICE engine is

(i) it cannot make any approximation to device models.
(ii) It tries to discretize every part of the circuit at uniform time.
(iii) It tries to iterate a huge matrix for global convergence.

This iteration and convergence in SPICE simulators are implemented with direct sparse solvers that generally has a time complexity of $O(N^2)$. This is a serious performance and capacity limitation to the SPICE simulator. With the emergence of large-scale digital memory circuits and mixed signal SOC full chip simulation, the number of elements in the input of a simulator reaches around a billion MOSFETS. For a simulation of that scale, a time complexity of $O(N^2)$ cannot be accepted. Hence relaxing accuracy constraints for performance becomes paramount. This is precisely what Fast-Spice aims at doing.

This tradeoff is pictorially depicted by the following:



Some key Fast-SPICE optimization techniques are

(i) Accurate device specific models are reduced into simpler tabularized models.
(ii) Event-driven simulation is performed, where solution in a given region is computed only if a change in state is anticipated due to a changing input or load.
(iii) Instead of global uniform time step, we have multi-rate simulation.
(iv) The circuit is partitioned into number of blocks, using carefully designed physical and mathematical criteria.
(v) Multithreading and parallel processing for improving solvers.

Fast-SPICE looks at circuit simulation in a comparatively different way than a conventional SPICE simulator. The three important views of a Fast-SPICE simulator are as follows:

(i) Matrix based
(ii) Graph based
(iii) Circuit based

In each of these views Fast-SPICE has implemented an optimized scheme when compared to a conventional SPICE simulator.

Matrix Based Views:

The Fast-SPICE simulation uses the following schemes:

(i) Effective pre-conditioners.
    i. Incomplete LU factorization
    ii. Multi-grid preconditioning
(ii) Optimized node ordering
(iii) Matrix partitioning
(iv) Multi-threading and parallel processing. For iteration and convergence.

Graph Based Viewpoint:

The whole electrical circuit is viewed as a graph. Each electrical component is one of the graph's edges. Connection between components is a node. Hence graph optimizations are used to reduce simulation-time and complexity.

Circuit-Based Viewpoint:

Whatever said and done, the application of the simulator is simulating an electrical circuit, Hence there are a lot of optimization done in the conceptual electrical domain. Electrical concepts are taken advantage off while doing effective pre-conditioning and dynamic partition.

Partitioning is an important part of Fast-SPICE simulation technology, by static or dynamic partitioning we break a big complicated circuit into its corresponding blocks and the blocks are iterated independently at different rates(multi-rates) during simulation.

Finally memory circuits (including DRAM,SRAM,FLASH and so on) are very large in size and follow regular topology. Very large portions of the memory array remains latent at a given point in time. These observations can be employed to develop optimization schemes that are very specific to memory circuits.

Customsim is based on Fast-SPICE technology and has many of the ideas discussed in the previous sections. On the architectural over view of Customsim, please refer to the following section.

For more detailed discussion about Fast-SPICE technology. Please refer to sources [4].

Section 3: Customsim Architecture:

Customsim has 3 main components, they are

(i) The Front end parser
(ii) DC engine
(iii) Transient engine

The parser parses the input netlist. Different types of netlists from various popular EDA tools are supported by the simulator. This netlist is parsed and it is stored in a database hence the engines can access the input data independently. The setup DC and setup transient does the necessary preconditioning. By preconditioning we access the database to construct a graph with each element in the electrical circuit as its nodes. Each an every node has a nodal property and each edge has an edge property. Once this graph has been constructed, The respective engines perform their respective analysis to give out the simulation results.

The simulator can be configured in such a way that the user has a say in the performance Vs. accuracy tradeoff. The simulator can be configured to operate in levels 0 to 7. This can be achieved using the option. "`set_sim_level <accuracy level>`" . Increase in levels means increasing accuracy. More and more simulations are done in a conservative manner. Other simulations features can also be activated by configuring the respective options in command line.

We shall talk about a few specific commands in the later part of the document for reasoning out various observations of our experiments.

Section 4: Our objective.

As a part of the internship program we set out to interface the simulator source code with the Synopsys memory allocator (MEM) and evaluate the performance of the simulator with the memory module. We have a benchmark binary that does not employ the memory module. We plan on comparing the two binaries some test cases and reason out our observations.

Section 5: SYNOPSYS memory allocator(MEM):

MEM is a Customsim memory allocator used in many Synopsys tools. Freespace management and coalescing are the most important attribute to look for in any memory allocator. Having said that, In my view, Independent performance of a memory allocator is almost of negligible significance. It is the performance of the application that matters. Hence important decisions about various attributes of the allocator is mostly very subjective. One must have a thorough understanding of the data access pattern of the application for which one is writing this memory module.

Two main parameters to look forward to while assessing the stand alone performance of the memory allocator is as follows:

(i) Utilization
(ii) Throughput

Utilization is how effectively a storage location is used. In our context it means, if asked for 'x' number of bytes any memory allocator uses 'x+Δx' number of bytes. The Δx is called metadata and is used to keep track of free spaces and other overheads from the memory allocator.

Throughput is about of memory allocated per unit of time. We have some information to have a vague idea from which we can infer about these parameters of MEM relative to a general GNU memory allocator.

Need for MEM:

The GNU malloc is of the time complexity $O(n^2)$ where n is the size of the free list. Thread-safe malloc is even more un-optimized and uses spinlocks. Spinlocks waste our computing resources and can have an alternate. GNU memory allocator remains that way due to portability and generality requirements.

Technical specification of MEM:

Page size in MEM is 8K. It is larger than GNU memory allocator.

Freespace management in MEM is done by using a global free list and AVL trees. Small allocations are allocated from the free list. Larger allocations are managed by the tree.

It has a 2-level page table and employs binning. It also has a small cache for each bin and allocates memory in groups. Hence if exactly one page of memory is freed, that page will get cached. When the next request for allocation comes, If it is less than or equal to the page size, it gets allocated immediately. This is advantageous as it eliminates the overhead of iterating of the free-list and also sorts out the page flutter problem.

This idea is also implemented in 'tcmalloc', the google memory allocator.

Allocating memory in groups also adds one more level of locality to the memory allocator.

Thread-safe memory allocation:

The GNU memory allocator uses spin lock to make its malloc thread safe. The issue with spin locks is unless the critical resource becomes free, the threads that are in contention keep polling to check if the resource is free. The operating system ‘s default scheduler(which is ‘Completely fair scheduler’ for Red Hat) gives equal preference to all the threads that are in contention, wasting a lot of computing resources. But MEM uses wait locks, which means the threads that are in contention and cannot acquire a resource in the first attempt, are put to sleep. Hence saving upon a lot of cycles. For more information about spinlocks refer Sources [5].

Performance when compared to GNU memory allocator:

		GNU memory allocator	MEM
Normal	Malloc	33 nsec	10 nsec
	Free	144 nsec	46 nsec
Thread safe	Malloc	78 nsec	37 nsec
	Free	188 nsec	78 nsec

Section 6: Implementation Issues.

Initially we tried intercepting all the ‘malloc’ calls in our application using ‘malloc_hook’. Malloc_hook is a system pointer that stores the address of where the ‘malloc’ function is stored. Hence whenever ‘malloc’ is called, wherever malloc_hook points that routine is supposed to run. But another technique is redefining ‘malloc’ in our header files. This redefinition is there in the #include "osi.h", #include "mem.h" . We have to call the ‘mem_initialize()’ function in our ‘main()’ function. If we do this, we successfully redefine ‘malloc()’ by’ mem_malloc()’. The ‘mem_initiailize()’ sets up other infrastructure of the memory allocator like page tables.

If we want to use other features of MEM we need add other header files. This might cause some redefinitions of some variables that are already in our source code. These dependencies has to be resolved. The MEM debug information also needs some environment variables to be initialized.

Section 7: Observation

After interfacing the MEM with our source-code, we ran our binary and a benchmark binary in 2 different grid architectures,

(i) SYNOPSYS GRID
(ii) NUMA GRID

Some architectural information about these grids are discussed in the analysis section. Most of the information we need for performance evaluation is a part of the 'log' file generated as an output.

The observations are of 4 tables.

Table 1: is test cases run on Synopsys grid.

Table 2: is test cases run on Numa grid.

Table 3: is multi-core test cases run on Numa grid.

Table 4: is test cases run with specific commands.

SYNOPSYS GRID : SINGLE CORE

TABLE 1:

Single Core	BM				MEM						
Testcase	FE	DC	TR	Total wall Time	FE	DC	TR	Total Wall Time	CPU model	Cores	Operating system
short normal	12	0	386	398	11	0	355	366	Xeon(R) CPU E5-2690 v2 @ 3.00GHz	12	Linux- 2.6.32-358.el6.x86_64
short-SRAM1	69	1	97	167	71	1	96	168	Xeon(R) CPU E5-2670 0 @ 2.60GHz	16	Linux - 2.6.32-358.el6.x86_64
short-SRAM2	31	3	140	174	30	3	140	172	Xeon(R) CPU E5-2643 v2 @ 3.50GHz	12	Linux - 2.6.32-358.el6.x86_64
medium-normal-1	32	0	1696	1728	26	0	1699	1725	Xeon(R) CPU E5-2670 0 @ 2.60GHz	16	Linux - 3.10.0-229.el7.x86_64
medium-normal-2	18	0	3448	3466	19	0	3435	3454	Xeon(R) CPU E5-2670 0 @ 2.60GHz	16	Linux-2.6.32-220.el6.x86_64
medium-SRAM1	121	22	1189	1332	122	22	1182	1326	Xeon(R) CPU E5-2670 0 @ 2.60GHz	16	Linux-3.10.0-229.el7.x86_64
medium-SRAM2	255	21	899	1175	252	20	897	1168	Xeon(R) CPU E5-2670 0 @ 2.60GHz	16	Linux-3.10.0-229.el7.x86_64

NUMAGRID: SINGLE-CORE

TABLE 2:

		BM				MEM					
Testcase Name	FE	DC	TR	Total Wall time	FE	DC	TR	Total Wall Time	CPU model	number of CPUs	operating system
short normal	12	0	426	438	11	1	369	381	Xeon(R) CPU E5-2690 v2 @ 3.00GHz	20	Linux - 2.6.32-358.el6.x86_64
short-SRAM1	64	2	86	152	63	2	86	151	Xeon(R) CPU E5-2690 v2 @ 3.00GHz	20	Linux - 2.6.32-358.el6.x86_64
short-SRAM2	32	2	151	184	31	3	148	182	Xeon(R) CPU E5-2690 v2 @ 3.00GHz	20	Linux - 2.6.32-358.el6.x86_64
medium-normal-1	12	0	1459	1471	13	0	1460	1473	Xeon(R) CPU E5-2690 v2 @ 3.00GHz	20	Linux - 2.6.32-358.el6.x86_64
medium-normal-2	17	0	3151	3168	17	0	3142	3159	Xeon(R) CPU E5-2690 v2 @ 3.00GHz	20	Linux - 2.6.32-358.el6.x86_64
medium-SRAM1	108	18	1071	1197	109	19	1096	1124	Xeon(R) CPU E5-2690 v2 @ 3.00GHz	20	Linux - 2.6.32-358.el6.x86_64
medium-SRAM2	224	17	811	1072	238	16	800	1054	Xeon(R) CPU E5-2690 v2 @ 3.00GHz	20	Linux - 2.6.32-358.el6.x86_64

NUMA GRID: MULTI-CORE

TABLE 3:

	BM				MEM						
Testcase Name	FE	DC	TR	Total Wall Time	FE	DC	TR	Total Wall Time	CPU model	number of CPUs	operating system
short normal	11	0	277	288	11	0	269	280	Xeon(R) CPU E5-2690 v2 @ 3.00GHz	20	Linux - 2.6.32-358.el6.x86_64
short-SRAM1	63	2	49	114	63	2	48	113	Xeon(R) CPU E5-2690 v2 @ 3.00GHz	20	Linux - 2.6.32-358.el6.x86_64
short-SRAM2	31	3	149	183	32	3	149	184	Xeon(R) CPU E5-2690 v2 @ 3.00GHz	20	Linux - 2.6.32-358.el6.x86_64
medium-normal-1	12	0	920	932	13	0	1653	1666	Xeon(R) CPU E5-2690 v2 @ 3.00GHz	20	Linux - 2.6.32-358.el6.x86_64
medium-normal-2	18	0	1472	1490	18	0	1452	1470	Xeon(R) CPU E5-2690 v2 @ 3.00GHz	20	Linux - 2.6.32-358.el6.x86_64
medium-SRAM1	108	18	515	641	106	18	521	645	Xeon(R) CPU E5-2690 v2 @ 3.00GHz	20	Linux - 2.6.32-358.el6.x86_64
medium-SRAM2	235	17	372	623	238	17	374	629	Xeon(R) CPU E5-2690 v2 @ 3.00GHz	20	Linux - 2.6.32-358.el6.x86_64

NUMAGRID: WITH OPTIONS

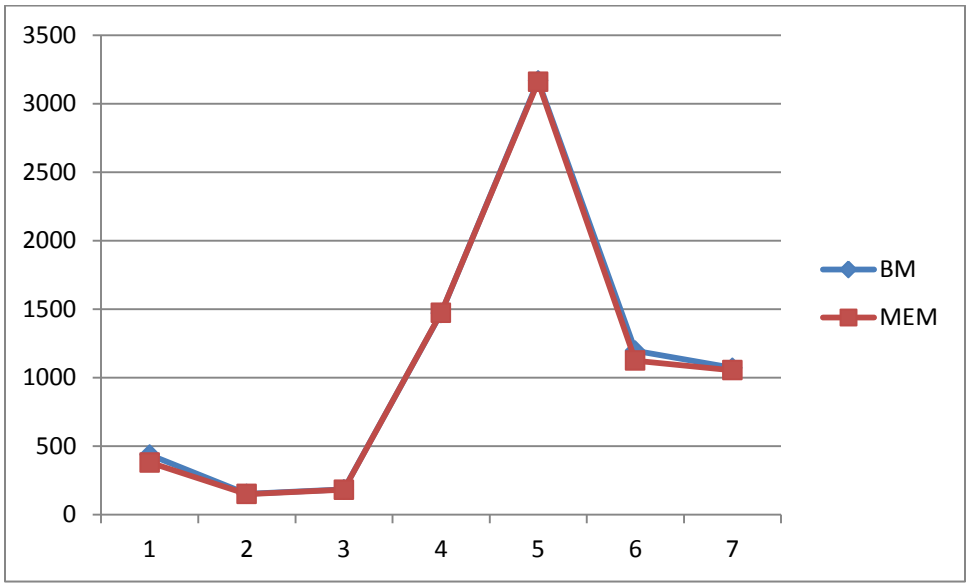
TABLE 4:

	BM				MEM						
Testcase Name	FE	DC	TR	Total Wall Time	FE	DC	TR	Total Wall Time	CPU model	number of CPUs	operating system
short normal	12	0	882	897	11	1	885	897	Xeon(R) CPU E5-2690 v2 @ 3.00GHz	20	Linux - 2.6.32-358.el6.x86_64
short-SRAM1	73	2	736	811	73	2	751	826	Xeon(R) CPU E5-2690 v2 @ 3.00GHz	20	Linux - 2.6.32-358.el6.x86_64
short-SRAM2	37	3	1391	1431	37	3	1398	1438	Xeon(R) CPU E5-2690 v2 @ 3.00GHz	20	Linux - 2.6.32-358.el6.x86_64
medium-normal-1	13	0	3185	3198	13	0	3135	3149	Xeon(R) CPU E5-2690 v2 @ 3.00GHz	20	Linux - 2.6.32-358.el6.x86_64
medium-normal-2	46	0	10626	10672	17	0	10794	10811	Xeon(R) CPU E5-2690 v2 @ 3.00GHz	20	Linux - 2.6.32-358.el6.x86_64

Section 8: Results/Graphs:

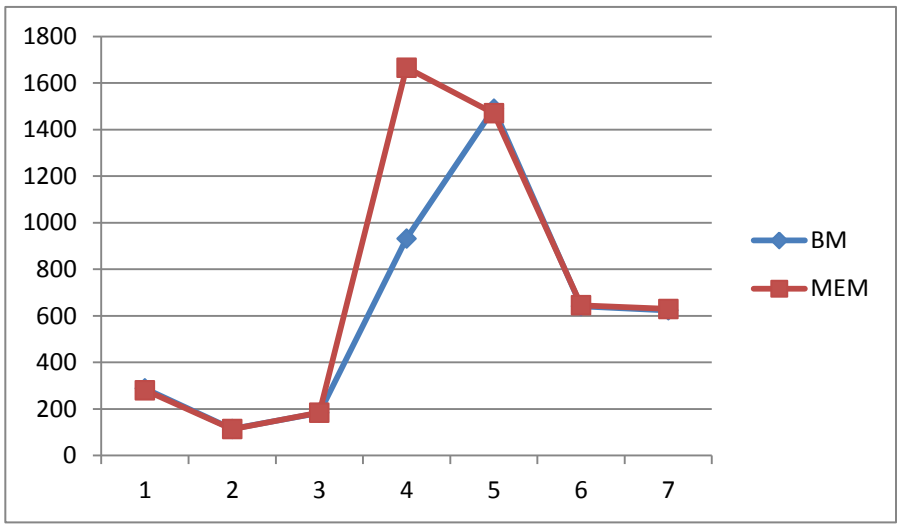
Table 2: Performance of benchmark vs MEM in a single core mode run on NUMA grid.

Wall times are plotted against one another.



S.no	BM	MEM
1	438	381
2	152	151
3	184	182
4	1471	1473
5	3168	3159
6	1197	1124
7	1072	1054

Table 3: Numagrid Multi-core, Total wall times plotted against each other.



S.no	BM	MEM
1	288	280
2	114	113
3	183	184
4	932	1666
5	1490	1470
6	641	645
7	623	629

Final comments: The performance of the application largely remains the same. There are portions of the simulator that has some gain associated to it but lack of performance in other parts nullify it. The only multi-threaded part of the simulator is the transient engine. Hence there is a marginal gain in its performance in the few of its test cases.

Another noteworthy observation is that when these two additional commands are added to the runs:

cck_analog_pdown –label floatdcpath –idsth 30n –rule 1 –HiZgate 1

cck_excess_ipath -label stby_leak -ith 10n -tth 20ps -tstep 1n

The simulator hence takes more time and memory. When run with some commands the front end parser of the simulator calls for more than the prescribed number of mallocs. Ideally a section of the simulator is supposed to allocate memory in an optimal manner(one large malloc-all subsequent malloc of that section has to redistributed and accommodated within this large malloc).

Section 9: Future work to be done:

(i) The part of the application that asks for memory in an non-optimal manner has to be fixed in order to expect an improvement in performance. This part of the simulator is from the 'lex' and 'yacc' files. These are a part of the front end parser.

(ii) Benchmarks to test the performance of stand-alone MEM. We have to iteratively try and allocate and de-allocate memory to look for MEM's throughput and utilization.

(iii) MEM's potential largely remains untapped. There are parts of MEM that when interfaced and used can bring about considerable improvement. We can interface them, resolve header file dependencies and hope for a performance improvement.

(iv) We can also tap into the debug information provided by MEM which can give us more insight on how each part of our simulator uses memory.

Sources:

1. The adjoining table[1] gives some of the popular EDA tools that use FAST-spice technology. -> https://www.semiwiki.com/forum/showwiki.php?title=Semi+Wiki:SPICE+and+FastSPICE+Vendors+Wiki
2. Basics of SPICE technology -> Kielkowski, Ron M. *Inside Spice*. Vol. 2. New York: McGraw-Hill, 1998.
3. Linearization in calculus -> https://en.wikipedia.org/wiki/Linearization ; https://www.khanacademy.org/math/differential-calculus/derivative_applications/local-linearization/v/local-linearization-intro
4. Rewieński, Michał. "A perspective on fast-SPICE simulation technology."*Simulation and Verification of Electronic and Biological Systems*. Springer Netherlands, 2011. 23-42.
5. http://stackoverflow.com/questions/5869825/when-should-one-use-a-spinlock-instead-of-mutex