Application Configurable DRAM Memory Management for HPC and Big Data

By Earle Jennings

Abstract—**DRAM operation is a major cost for HPC and big data programs. Also, DRAM has hard error reliability problems. This paper proposes how to minimize DRAM energy consumption and increase reliability. Performance is also maximized, and the needs of exascale systems, which require local system state capture and rollback, are met. Three aspects of DRAM application specific memory management are discussed. First, each node requires a specific amount of DRAM. A node is the minimum hardware component needed to execute the application. Second, entire system applications often require many nodes using DRAM, for example as parallel processors. Third, in sparse matrix simulations of partial differential equations, the implementation is organized to make the mathematics transparent, not to optimize system performance. The benchmark HPCG is used to show this. A multi-tiered approach to managing and operating DRAMs is proposed. First, logical to physical memory address conversion can minimize error impact, and determine an application's DRAM allocation. Second, this allocation leads to only powering the needed DRAM chips, rather than all of them. The logical addressing of the operating system can remain stable, while the physical placement changes over time. This minimizes the wear on the DRAM chips. Third, extending the logical addressing to encompass large data spaces is discussed. Fourth, the addressing scheme is extended to support distributed data objects in terms of their defined relationships. Again, this is illustrated in terms of HPCG.**

Keywords— DRAM; memory management; codesign; HPC; big data; exascale; HPCG; GRAPH 500

I. Introduction

DRAM is the only available storage technology able to handle the main memory requirements of numerical, or data, intensive computations. DRAM can be reliably manufactured in chips storing billions of bits, and be reliably written trillions of times. The nearest potential alternatives either cannot be reliably manufactured to hold comparable amounts of data, or cannot be reliably written more than a few tens of thousands of times. However, the energy cost of operating the DRAM is a major component of the cost of program execution [2]. DRAM also faces hard error problems [1]. Damaged DRAM components must be replaced, contributing a significant maintenance expense. Today, DRAMs are controlled by memory controllers very similar in structure to those introduced in the 1970's. These controllers keep all DRAMs always turned on, irrespective of how much memory is required by an application (app). By the 1990's, the DRAM controllers were slaved to a cache, which is interfaced to one or more microprocessors. These caches request a page of 64, or more bytes, from the DRAM. This leads to access problems for sparse matrix solvers, where only a small part of the requested page contents are actually used. HPCG [3] is a computer benchmark which highlights this problem. While there are many studies of DRAM problems and potential solutions [6] to [17], this article focuses on "Cosmic rays don't strike twice: understanding the nature of DRAM errors and the implications for system design" [1]. It is a statistically relevant survey, and provides a clear set of observations from which this approach can be derived and understood.

This paper proposes a co-design approach to managing DRAM operations. Hardware and software work together to reduce DRAM reliability and communication problems, as well as memory access overhead. This approach uniquely configures DRAM for each application, at each application node, dramatically reducing DRAM usage and energy consumption. This approach is applicable across most, if not all, hardware approaches requiring DRAMS.

The proposed circuitry is applied in this paper to two example apps, both using the same nodes, with dramatically different DRAM requirements. The first app performs data mining, which requires 256 Hadoop nodes, each using 64 Gbytes of DRAM, for a total amount of DRAM of 16 Terra bytes.

The second app implements a sparse matrix solver of a system of Partial Differential Equations (PDEs) similar to the HPCG benchmark [3]. This app uses a total of 157 Gbytes over all the nodes. However, it triggers frequent cache misses, which cripple performance. This app uses the finite difference scheme of HPCG on a 280 by 320 by 540 grid with 48,384,000 grid points. This simulation is represented by a sparse system of linear equations, with a typical stencil of 27 grid points away from the grid boundary, shown in Fig. 1. The stencil operator is shown in a three dimensional (3-D) coordinate system. The stencil operator operates upon the nearest neighbors of the point at '0', which is the point in this grid being approximated.

The linear system is modeled by a sparse matrix A, a stimulus vector x and a resulting vector b, where the solution satisfies $Ax = b$. Suppose that an implementation uses three forms of the stimulus vector and three forms of the response vector. Together with the sparse matrix A, there are seven global objects, which are distributed across the nodes to implement the simulation. Assume that each grid point includes a number (double precision floating point) and indexing of 4 bytes to indicate the column index of the grid point. It is common to organize the A matrix into row-compressed notation, so that all the elements of a given row are

stored in consecutive locations. Each entry of the matrix A contains about 12 bytes for each grid point coefficient and its column index. Given this entry size, the matrix contains about 12*27*48,384,000 bytes, which is about 157 Gbytes. Again assume that 256 nodes are being used, each node need only contain about 1 Gbyte.

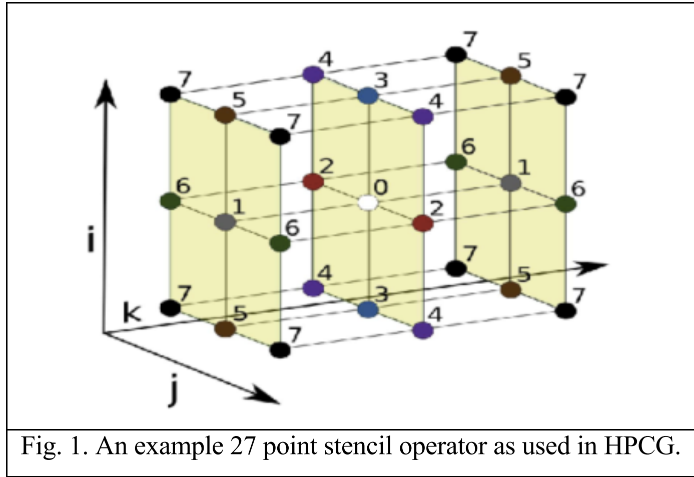

Fig. 1. An example 27 point stencil operator as used in HPCG.

HPCG is a typically sized grid of 280 by 320 by 540 nodes [3]. However, the sparse linear system of equations frequently codifies these models as shown in Fig. 2. The index location (x,y,z) = (z*N2+y)*N1+x is used to encode the indexing of the stencil into the sparse matrix and vectors of the model of HPCG. Here, N1 is 280 and N2 is 320 so that N1*N2 = 280*320 = 89,600 double precision numbers. The Euclidean distance between L0 and L1 = 1, whereas the difference between index locations L0 and L1 is N1*N2. This figure shows that the nodes of this grid are enumerated first in one direction, for instance, along the i-axis, then along the j-axis, and then along k-axis. This scatters the geometric locations across a wide section of the stimulus vector components.

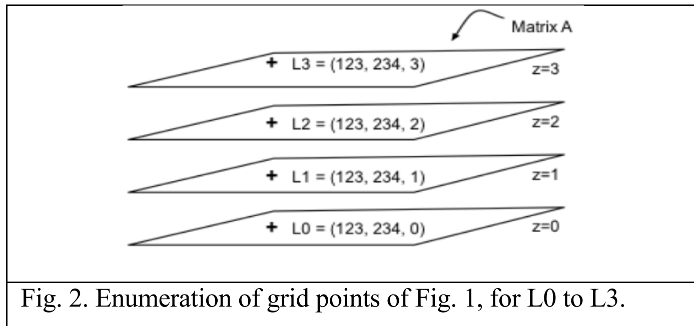

Fig. 2. Enumeration of grid points of Fig. 1, for L0 to L3.

Consider the two neighboring points L0 and L1 of Fig. 2. The Euclidean distance between L0 and L1 is 1, but distance between their index locations in the corresponding stimulus vector components, is 280*320 = 89,600 double precision numbers, or 716,800 bytes apart from each other. Note that two points separated by one in the j direction are 2,240 bytes apart. If two points are separated by two numbers in the j direction, this is enough to trigger a cache fault when the cache page size is 4096 bytes. Caching structures force a block of memory, often at least 4K bytes in size to be fetched for the each of the 8 byte floating point numbers required to calculate based upon one of these stencils. To access the z-0, 1 and 2 planes for one 27 point stencil often entails at least four data transfers to read 9 of the 27 double precision FP numbers for a new value of z. This is a communication cost of 4*4096 bytes to access 216 bytes, which is more than 97% wasted.

The first app seems to be a more demanding task, because of its much larger demand of DRAM memory. However, the Hadoop-compliant nodes of the first app are almost exclusively processing just the data at their node. The caching of the Hadoop node's local DRAM is unlikely to have cache faults outside the node's DRAM. The second, HPCG app forces cache misses and fetches from outside the node almost all the time.

Today, the DRAM is controlled by memory controllers, which are not configured to optimize the app running on the node. A node is the basic execution unit of an app's programs and configurations, each applied to internal resource(s) in one or more of the node(s). For example, a program may instruct a microprocessor directing multiple SIMD arrays of VLIW cores, such as the Sunway computer employs [19]. A program may instruct the operations of a Graphical Processing Unit (GPU). A configuration may direct one or more FPGA(s) to operate as an accelerator for numeric and/or data intensive computations. Within this context, the DRAM is operated for use by a node's internal resources as in Fig. 3.

This paper introduces an app configurable DRAM controller, which can optimize the app's execution on the node, and with other app nodes, throughout the system. The app configurable DRAM controller interfaces with a collection of DRAM chips, called a DRAM chip array. The DRAM chips are organized into chip units, which can be allocated to operate, or be held in reserve. The operational chip units are powered during the app's execution. The reserve chip units are not normally powered during the app's execution. This application configuration of the node insures that only the chip units required by the program are consuming energy. Each app node can have internal resources, as well as an app configurable DRAM controller interfaced to a DRAM chip array organized into chip units of one or more DRAM chips. Each DRAM chip includes multiple pages organized around one or more rows of the DRAM. The pages, which are what is actually accessed, traverse multiple columns.

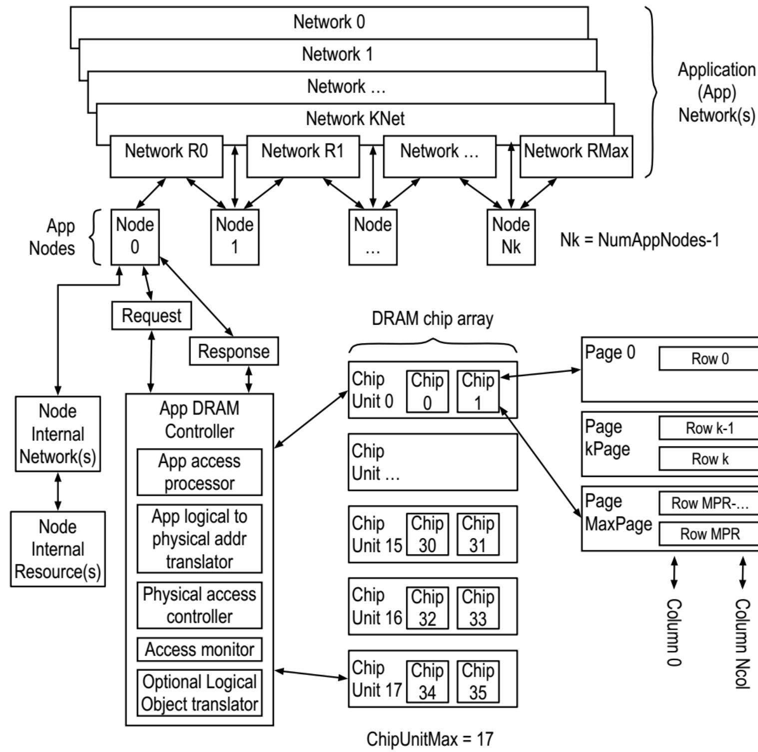

Fig. 3. A system with one or more application (app) nodes, communicating via one or more network(s).

II. Logical to Physical Addressing

There are two persistent, but relatively small, data requirements in these nodes, which may vary from one app to another. First, the operating system kernel will have a memory component. Second, nodes need to support local, resilient, response to exascale system failures, which will be discussed at the end of this section.

DRAM failure maps [1] indicate that specific areas receive a disproportionate amount of failures. There is a powerful, though implicit, consequence to these observations. A local to physical address map can be constructed, which averages out access usage of the physical pages of the DRAM. The logical to physical address map is also used to retire pages before they fail. The app logical to physical address translator uses the logical to physical address map. The logical to physical address translator is the first major component of the app configurable DRAM controller of Fig. 3.

The access processor receives requests for DRAM access, and uses the logical to physical address translator to direct the physical access controller to perform the accesses, when they are local. An access monitor continuously senses these local DRAM accesses to update the chip unit status tables in Fig. 4.

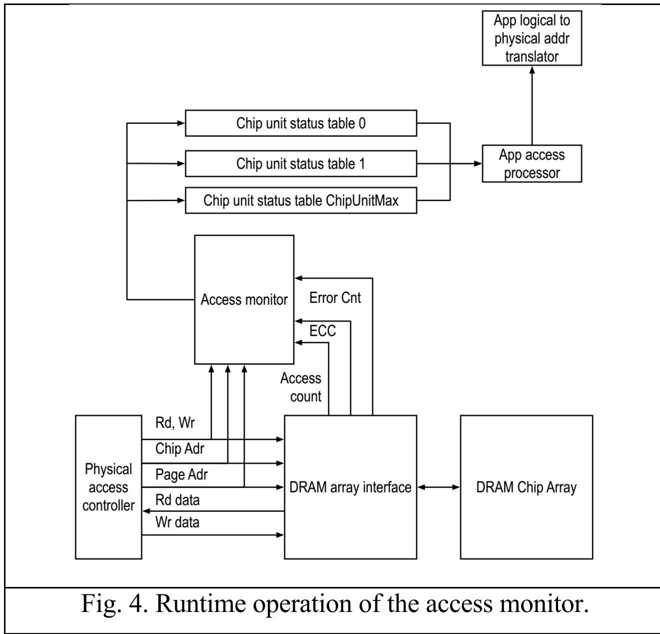

Fig. 4. Runtime operation of the access monitor.

The chip unit status tables are used by the access processor to reconfigure the logical to physical translation, and to optimize energy use for the app, as shown in Fig. 5.

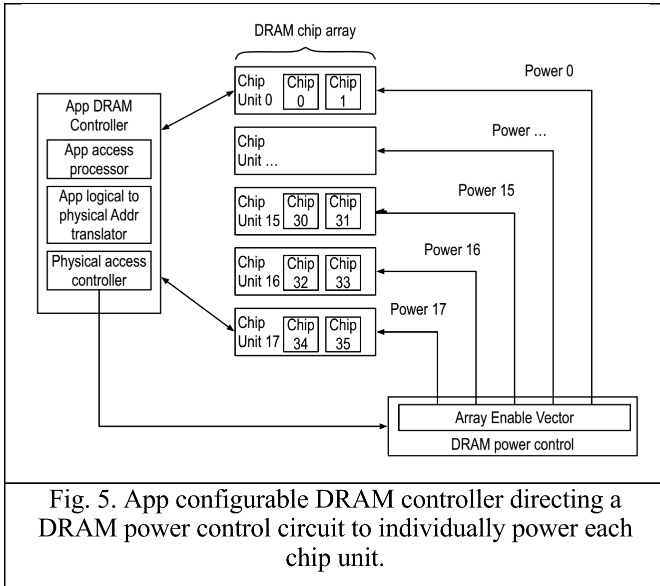

Fig. 5. App configurable DRAM controller directing a DRAM power control circuit to individually power each chip unit.

Given the variance in application DRAM requirements, energy conservation issues, app access requirements, and support for exascale class state resilience, we can formulate what needs to be logged at the app DRAM memory controller. This logged information guides an app specific mapping of logical to physical addresses at each node, and across all the nodes of the application system. The access monitor counts read and write accesses, as well as errors that can be corrected. The logs allow access balancing for frequently used memory regions when the logical to physical address map is generated. However, there is a third element to this situation, the application's usage of the DRAM.

Consider initializing DRAM access for the application throughout its application nodes in the system. Each app node knows its node address in a node total address space, its local DRAM requirement in terms of the total DRAM requirement of the application throughout the system, as well as a map of the local operating system kernel in the active DRAM of the node. Given this initialization information, several preliminary calculations used in building the logical to physical map for the application can be carried out with the above information, at each node simultaneously. The total DRAM requirement is used to create the application logical address space in terms of a starting block address, probably 0 up to a top block address offset in terms of the total DRAM requirement.

The app DRAM controller operates the active chip units. Fig. 5 shows some details of Fig. 3, in which the DRAM chip array is powered by a DRAM power control circuit directed by the physical access controller of the app DRAM controller. The DRAM power control may be included in the app DRAM controller, or it may be separately packaged. Reserve DRAM chip units are not powered up during the execution of the app. The operational chip units are powered during the execution of the application program. This app configuration of the DRAM controller insures that only the chip units required by the program are consuming energy, which minimizes DRAM energy use.

Returning to Fig. 3, the app access processor receives access requests, and operates the other elements of the app DRAM controller to respond to the access request. Here are some examples of these activities as two read requests: The app access processor receives a first read request for DRAM at a given logical address. The logical address may indicate the corresponding physical address is located within the app node's DRAM array. In this situation, the app access processor queries the logical to physical address translator to determine one, or more, physical addresses required by the read request. The physical address(s), plus possibly an access length, are provided to the physical access controller, which is instructed by the app access processor to read the appropriate physical page(s) of the DRAM array to perform the requested read operation. The contents of the DRAM are sent to the app node as the appropriate response, when the read operation has either been without error, or has been corrected appropriately.

The app access processor receives a second read request at a second logical address, which corresponds to a physical address outside the DRAM array of this app node. The app access processor responds by sending an access request message across one, or more, app networks to another app node, to fetch the requested DRAM memory contents.

Consider how to minimize the overhead associated with the HPCG benchmark, and sparse matrix manipulation, and other apps, such as graph related apps as demonstrated by the Graph 500

benchmark [18]. The intent of the app programmer is to access the large data objects at will, by referencing where in the large object the access is to occur. They need the referenced data, but nothing more than that. This is where the logical object translator of Fig. 1 enters into the picture. The logical object translator receives a logical object identifier and an internal index list, which is processed to create a local indicator and a non-local indicator. The local indicator is true when at least part of the object access request can be processed locally. The non-local indicator is true when at least part of the object access request must be processed away from the app node. The response to these requests is the transfer and access of just the information required for the transfer. This approach, by using the appropriate networks, entails some overhead, but does not trigger the access and transfer of thousands of unneeded bytes, as a caching system requires.

Now consider some of the details of Fig. 4. The physical access controller performs DRAM accesses, by communicating to a DRAM array interface, which directly accesses one or more of the active Chip Units in the DRAM Chip Array. The following are communicated: Read (Rd) and Write (Wr) signaling, combined with a Chip address (Adr), Page address (Adr). Write operations assert the Wr data from the physical access controller to control the DRAM array interface to write into the DRAM Chip Array. Read operations find the DRAM array interface asserting the Rd Data in response to its read access of the DRAM chip array.

The access monitor intercepts these signals and an Access count, an Error Correction Code (ECC) status report, and an Error count (cnt) to create an update of one or more of the Chip unit status tables 0 to ChipUnitMax, which is 17 in the current example. Note that the ECC coding associated with the Wr data for write operations may be generated in the physical access controller or the DRAM array interface. The ECC coding associated with the Rd data for read operations is generated by the DRAM array interface in response to the DRAM chip array being read. The DRAM array interface analyzes the Rd data and Rd ECC (not shown) from the DRAM chip array to determine if the read is without errors, or if the errors are detected but correctable, or if the errors are detectable and uncorrectable. These various conditions make up the ECC access status report, as well as the access count for the access operation to the access monitor.

The access monitor organizes the inputs shown in Fig. 4, and updates one or more of the chip unit status tables. These chip unit status tables are retained to monitor the wear on the DRAM chip array. The tables may have an essentially non-volatile memory, and/or may be transferred elsewhere for archiving. Note that in some situations, these tables may be merged and potentially organized in a different fashion, but provide comparable, equivalent or more detailed information about the wear of the DRAM components, possibly in support of one or more page retirement policies [1]. These page retirement policies are shown to remove 90% or more of the page failures. The domain of each design and its system requirements, determines the structure and behavior of the DRAM chips being used, and leads to decisions about which specific policy is to be implemented.

While the DRAMs may support some form of multi-block operations or structure in each of their chips, this discussion focuses on the simplest model of their behavior that is likely to apply. So while a chip status table may, or may not, exhibit determinations of a block structure, the following is more certain to be useful: Each chip unit table includes an array of PageUsageSummary, possibly indexed to traverse the pages of the chip unit. Each PageUsageSummary includes the following indicators of usage of the page being referenced:

- NumberOfWrites approximates how many writes have been performed on the page,
- NumberOfReads approximates how many reads have been performed on the page,
- a PageStatus, may indicate some combination of UseOrUnused, possibly in conjunction with CanOrCannotBeUsed,
- a NumberOfCorrectableErrors,
- a CorrectableErrorAddressList, and
- a Time2FailureEstimate.

```
TotalActivePages = 0
Do
   (NewChipUnitID, NewActivePages) = CalculateMaxAndRemove(
      TotalAcceptableActivePagesList)
    AddActivePhysicalPagesToLogic2PhysicalAddr_Table( NewChipUnitID,
      NewActivePages)
    Assert Used(NewChipUnitID) in UnitPageSummary of every acceptable page
        In the NewChipUnitID
    TotalActivePages = TotalActivePages + NewActivePages
While TotalActivePages < TotalLogicalActivePageCount
```

Above is Listing One, showing an example of an allocation procedure for chip units. This example minimizes energy consumption by allocating the most usable chip units first. This also insures a high Mean Time Between Failure (MTBF), by

allocating chip units with the highest number as new active pages. Each of these pages has an acceptably large estimate for the time to DRAM page hard failures.

The app access processor may initialize the app logical to physical address translator by first initializing the local logical address table with TotalLogicalActivePageCount, which is the logical address field range of the app node multiplied by the ideal number of pages for the logical chip unit. Either the app access processor or the access monitor may operate upon the chip unit status table to update the Time2FailureEstimate of each page of each chip unit in the DRAM array.

The app access processor may then begin to build a new active chip unit table. The first step is to calculate for each of the chip units, how many pages of the chip unit have acceptable Time2FailureEstimates, which will be referred to as the total acceptable active pages, for each chip unit. Upon completing the activities outlined in Listing One, the app node DRAM is ready to be initialized with data, which is written as indicated by its logical address into pages as indicated by its corresponding physical address sent from the logical to physical address translator.

Exascale requires periodic saving and restoring of the internal state of nodes due to system faults. Consider node 0 of Fig. 3, which has some number of internal resources. These internal resources, no matter what their internal architecture, will tend to have a data related internal state. Assume the states of the internal resource are at least 20 Mbytes in size. The most immediately available, fault tolerant memory to these resources is the DRAM chip array. Assuming that the maximum standard, main memory to be the Hadoop single window of 64 Gbytes, and that 10% additional pages are held in reserve to address page faults and balance the wear on the DRAM, the 18 chip units have 0.4 * 4 Gbytes available to act as the snapshot-rollback store for the node. To maximize the MTBF for all of the shown DRAM, this specialized memory allocation should also be part of the logical to physical map for the app. However, it is a separate allocation scheme, which is not accessible by the app itself. It works the same as the app's access processes. Also, when starting each app, the previous app's state should not be visible to the next, which can be assumed to be part of initializing these pages at the start of the app.

III. Summary

A new app configurable DRAM controller is discussed. This controller enables the use of a logical to physical address table. This table is used to translate the logical addressing into physical addressing of an application's main memory residing at one or more nodes. The logical to physical addressing insures that only the minimum number of DRAM chip units are powered, minimizing DRAM energy usage. Only the DRAM needed by an application are used. This logical to physical address translation supports evening the wear on the DRAM pages, as well as retirement of the DRAM pages that either have, or are likely to have, unrecoverable read failures. The app configurable DRAM controller also provides a logical object translator, which receives an access request for a logical object and an index referencing into the logical object's data. The logical object translator responds to this request by determining the relevant local and non-local activities that need to be performed, and signals the DRAM controller appropriately to insure the request is processed. The logical object translation and data message response add another mechanism besides the use of caching. Logical object access insures that only the required data is transferred between nodes. This provides a significant efficiency improvement for applications like sparse matrix manipulation as characterized by the HPCG benchmark.

Acknowledgment

The author wishes to thank Heather Murphree. If there is any clarity of presentation in this document, it undoubtedly stemmed from her questions and feedback. If there are any discrepancies or errors found herein, they are strictly the responsibility of the author.

REFERENCES

[1] A. Hwang, I. Stefanovici and B. Schroeder, “Cosmic rays don’t strike twice: understanding the nature of DRAM errors and the implications for system design”, ASPLOS’12, March 3–7, 2012, London, England, UK. Copyright 2012 ACM 978-1-4503-0759-8/12/03

[2] DOE ASCAC Subcommittee, Bob Lucas (Subcommittee Chair), “Top ten exascale research challenges”, Feb, 2014

[3] M. Heroux, J. Dongarra, P. Luszczek, “HPCG Benchmark Technical Specification”, Sandia National Laboratories Albuquerque, New Mexico 87185 and Livermore, California 94550, SAND-2013-8752, Released Oct, 2013.

[4] E. Jennings, “Core module optimizing PDE sparse matrix models with HPCG example”, July 25, 2017, pg 54-70, Supercomputing Frontiers and Innovations, vol. 4 no. 2, downloaded July 25, 2017 from http://superfri.org/superfri/issue/view/14, DOI: 10.14529/jsfi170205

[5] E. Jennings, “The Simultaneous Transmit And Receive (STAR) Message Protocol”, July 25, 2017, pg 38-53, vol. 4 no. 2, downloaded July 25, 2017 from http://superfri.org/superfri/issue/view/14, DOI: 10.14529/jsfi170204

[6] K. Cho, W. Kang, H. Cho, C. Lee, S. Kang, “A Survey of Repair Analysis Algorithms for Memories”, ACM Computing Surveys, Vol. 49, No. 3, Article 47, Publication date: October 2016, © 2016 ACM 0360-0300/2016/10-ART47, DOI: http://dx.doi.org/10.1145/2971481

[7] A. Bacchini, M. Rovatti, G. Furano, M. Ottavi, “Characterization of Data Retention Faults in DRAM Devices”, unknown publication data

[8] L. Chen, “Energy-efficient can cost-effective reliability design in memory systems”, (2014). Graduate Theses and Dissertations. Paper 13710. Downloaded from htp://lib.dr.iastate.edu/etd

[9] B. Giridhar, M. Cieslak, D. Duggal, R. Dreslinki, H. Chen, R. Patti, B. Hold, C. Chakrabarti, T. Mudge, D. Blaauw, “Exploring DRAM Organizations for Energy-Efficient and Resilient Exascale Memories”, © 2013 ACM 978-1-4503-2378-9/12/11

[10] A. Gainaru, Failure Avoidance Techniques for HPC Systems Based Upon Failure Prediction, PhD Dissertation, University of Illinois, Urbana-Champaign, 2015

[11] J. Dongarra, T. Herault, Y. Robert, “Fault tolerance technigues for high-performance computing”, May, 2015

[12] V. Sridharan, J. Stearley, N. DeBardeleben, S. Blanchard, S. Gurumurthi, “Feng Shui of Supercomputer Memory: Positional effects in DRAM and SRAM Faults”, SC 13 November 17-21, 20913, Denver, CO, USA, © 2013 ACM 978-1-4503-2378-9/13/11

[13] V. Sridharan, N. DrBardeleben, S. Blanchard, K. Ferreira, J. Stearley, J. Schalf, S. Gurumurthi, “Memory Errors in Modern Systems: The Goodm The Bad, and the Ugly”, ASPLOS 2015, March 14-18, 2015, Istanbul, Turkey, © 2015 ACM 978-1-4503,-2835-7/15/03

[14] J. Kim, Strong, Thorough, and Efficient Memory Protection against Existing and Emerging DRAM erros, PhD Thesis, University of Texas at Austin, Dec. 2016

[15] C. Costa, Y. Park, B. Rosenburg, C. Cher, K. Ryu, “A System Software Approach to Proactive Memory-Error Avoidance”, SC 2014 978-1-4799-5500-8/14 $31.00 © 2014 IEEE, DOI 10.1109/SC.2014.63

[16] A. Patwari, I. Laguna, M. Schulz, S. Bagchi, “Understanding the Spatial Characteristics of DRAM Errors in HPC Clusters”, FTXS’17, June 26, 2017, Washington , DC, USA, © 2017 Association for Computing Machinery, ACM ISBN 978-1-4503-5001-3/17/06, https://doi.org/http://dx.doi.org/10.1145/3086157.3086164

[17] P. Nair, ARCHITECTURAL TECHNIQUES TO ENABLE RELIABLE AND SCALABLE MEMORY SYSTEMS, PhD Dissertation Georgia Institute of RTechnology, May 2017, arXiv:1704.03991v1 [cs.AR]

[18] R. Murphy, K. Wheeler, B. Barrett, J. Ang, “Introducing the Graph 500”, Sandia National Laboratories, May 5, 2010.

[19] J. Dongarra, “Report on the Sunway TaihuLight System” June 20, 2016, University of Tennessee, Department of Electrical Engineering and Computer Science, Tech Report UT-EECS-16-742