Smart I/Os: A Data-pattern Aware 2.5D Interconnect with Space-Time Multiplexing

Sai Manoj P. D., Kanwen Wang, Hantao Huang and Hao Yu
School of Electrical and Electronic Engineering
Nanyang Technological University, Singapore 639798
Email: haoyu@ntu.edu.sg

ABSTRACT

A data-pattern aware smart I/O is introduced in this paper for 2.5D through-silicon interposer (TSI) interconnect based memory-logic integration. To match huge many-core bandwidth demand with limited supply of 2.5D I/O channels when accessing one shared memory, a space-time multiplexing based channel utilisation is developed inside the memory controller to reuse 2.5D I/O channels. Many cores are adaptively classified into clusters based on the bandwidth demand by space multiplexing to access the shared memory. Time multiplexing is then performed to schedule the cores in one cluster to occupy the supplied 2.5D I/O channels at different time-slots upon priority. The proposed smart 2.5D TSI I/O is verified by the system-level simulator with benchmarked workloads, which shows up to 58.85% bandwidth balancing and 11.90% QoS improvement.

1. INTRODUCTION

The existing many-core microprocessors with shared main memory integration [1] has limited bandwidth that is non-scalable for exa-scale computing. The 3D integration by through-silicon via (TSV) [2, 3, 4, 5] can significantly improve the memory-logic communication bandwidth but has severe thermal reliability concerns due to poor heat removal [6]. Recent development of 2.5D integration by through-silicon interposer (TSI) [7] is one promising solution by integrating multiple dies on one common substrate. It has good thermal dissipation as well as high bandwidth and low power when realized as the transmission line (T-line) deployed underneath. Compared to 2D PCB trace for memory-logic integration, 2.5D TSI I/O can provide better communication bandwidth. In contrast to 3D TSV I/O for memory-logic integration, the 2.5D TSI I/O has much better thermal reliability [8, 9].

The use of many-core microprocessor with shared memory for data-oriented commutation involves huge communication bandwidth demand. As there are limited number of TSI I/O channels to access the shared memory, this calls for a channel reutilization under quality-of-service (QoS) [10] constraint. Memory controller [11, 5] is commonly employed as the bridge between cores and shared memory. State-of-the-art memory controller [12] is mainly designed as off-chip interface with a large number of I/O pins for memory-logic integration to be managed. A better QoS can be achieved by the matching of memory-access demands by dynamic allocation of I/O channels and scheduling of requests [13]. A copious memory-access scheduling techniques are available in literature [13, 14, 15, 16, 17, 18]. Reordering of memory-access requests requires a considerable amount of modifications in the memory controller design. From the I/O perspective, memory channel partitioning (MCP) scheme for conventional memory is proposed [17], where the channels connecting memory and logic blocks are grouped based on the number of memory-access requests (MPKI) and the number of channels are determined based on the number of applications in the group. The utilization of bandwidth is not efficient because low memory applications have less memory-access demands and bandwidth of the assigned channels are under utilized. A batch scheduling technique is proposed in [14]. A limited number of requests are grouped as *batches* based on their arrival times and priorities are assigned based on the arrival times. What is more, in each batch, the requests are allowed to access the memory in parallel manner. This method provides fair allocation of channels, but the allocated bandwidth for parallel access may be wasted in case of low memory intensive applications. In [18], network-latency sensitive applications are mapped to the network (cores) running less bandwidth intensive applications such that the memory-access can be granted for latency sensitive applications. However, this technique has overhead in terms of application mapping and moreover, the channels are not balanced.

In this paper, we introduce a smart I/O, which is a 2.5D TSI I/O with space-time multiplexing based on the memory-access data-pattern characteristics to improve the channel utilization rate with good QoS. Different application workloads are first classified based on their memory-access data-patterns at different time-slots. Correspondingly, one can cluster the cores based on the demanded bandwidth characterized by the magnitude of memory-accesses (memory-access number). Accordingly, the I/O channels are allocated to different clusters, which is called *space multiplexing*. Moreover, inside one cluster, cores will be assigned with different priority characterized by the phase of memory-accesses. As such, cores in each cluster can be scheduled to occupy the allocated 2.5D I/O channels at different time-slots upon the priority, which is called *time multiplexing*. Such a space-time multiplexing improves the effective utilization of available 2.5D I/Os.

The proposed memory-access data-pattern aware smart with space-time multiplexing is implemented on a cycle-accurate simulator for up to 64-core microprocessor with shared memory, which can be explored in 2D

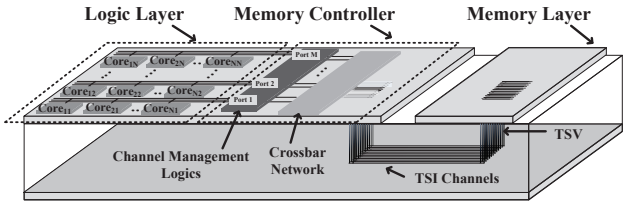

Figure 1: Side-view of 2.5D many-core microprocessor and main memory integration by TSI I/O channels.

integration as well. The memory intensive benchmarks from SPEC 2006 [19], PARSEC [20] and Phoenix [21] are selected. Experiment results show that the proposed memory controller can achieve up to 58.85% bandwidth balancing and 11.90% QoS improvement.

The remainder of the paper is organized as follows. Section 2 describes the system architecture with formulated problem of space-time multiplexing in the memory controller. Section 3 illustrates the study of memory-access data-pattern classification and QoS evaluation. The space-time multiplexing algorithm utilized for the smart I/O is presented in Section 4. Validated experiment results are discussed in Section 5 with conclusions drawn in Section 6.

2. 2.5D ARCHITECTURE OF SHARED MEMORY MANY-CORE MICROPROCESSORS

In this section, an architecture of 2.5D TSI integrated many-core microprocessors and a shared memory is presented with the focus on the data-pattern aware smart I/O for space-time multiplexing. A space-time multiplexing problem is formulated to achieve a demand-supply matching that can improve the I/O utilization rate with good QoS.

2.1 System Overview

Figure 1 shows the 2.5D architecture utilized for many-core memory-logic integration with the reconfigurable memory controller for 2.5D TSI I/O channel management. The die on the left composes of many-core microprocessors and the die on right is the shared main memory. Cores can access the main memory through a reconfigurable crossbar switch-network [22] to perform adaptive space-time multiplexing inside the memory controller by configuring data-pattern aware 2.5D TSI I/O channel connections. As TSIs are deployed underneath the common substrate, area overhead is mitigated. The crossbar switch-network is suitable for such a many-core microprocessor with shared memory by the following advantages: simple one-hop routing and ease of implementing QoS policy. Besides, it can be easily reconfigured based on the data-pattern analysis. The switch-network can be reconfigured to connect with 2.5D TSI I/O channels. In fact, one can model the 2.5D system architecture by a demand-supply system with the following three components:

- *I/O channel demander:* a set of microprocessor cores C of set-size N_c having demand for different bandwidth. Each core c_i is to have a bandwidth demand of $B_d(c_i)$.
- *I/O channel supplier:* N_{ch} TSI I/O channels have a total supply bandwidth B_T. The allocated bandwidth $B_a(c_i)$ for each core c_i is supplied with TSI I/O channels by the reconfigurable memory controller.
- *I/O channel controller:* a set of M memory ports with scheduler inside the reconfigurable memory controller to map requests from N_c cores to memory through N_{ch} TSI I/O channels under demand-supply matching by flexible crossbar switch-network.

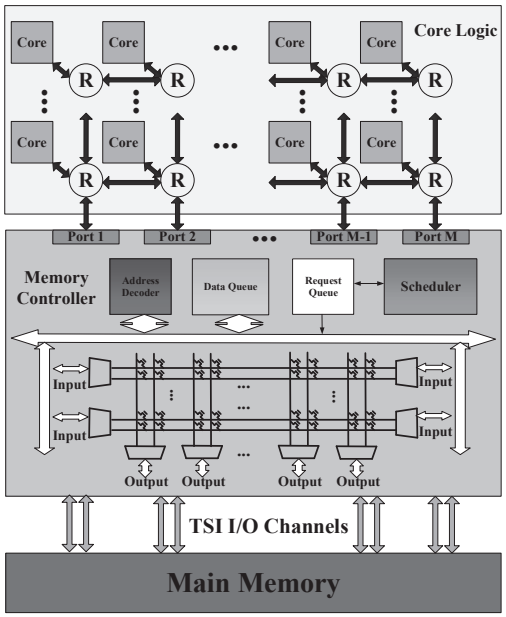

Figure 2: Memory-access data-pattern aware reconfigurable memory controller with space-time multiplexing.

Note that the TSI I/O channel management in memory controller can be implemented by simple logics of address decoder, data queue, request queue and scheduler.

2.2 Space-Time Multiplexing

The process of matching huge memory-access bandwidth demands by the data-pattern aware smart TSI I/O channels using the space-time multiplexing can be described as follows. Initially, based on the demanded bandwidth $B_d(c_i)$ from core c_i, the connection to one of the M ports of memory controller is established by on-chip routers. As such, cores with similar bandwidth demands form one cluster and connected to one port inside the memory controller. Note that each port can connect to any of the N_{ch} TSI channels by crossbar switch-network. A number of TSI I/O channels are further allocated to one port based on the demanded signature bandwidth, which is called as *space multiplexing*. What is more, as the cores with similar bandwidth demands can differ in memory-access priority, they occupy the channel at different time-slots, in a time multiplexed manner. The configuration is dynamically changed depending on the results of clustering at run time. Time multiplexing avoids overloading at a port due to allocation of similar workloads. This space-time multiplexing based smart I/O not only meets the demand, but also improves QoS with better I/O utilization rate. A more detailed view of the proposed architecture is presented in Figure 2.

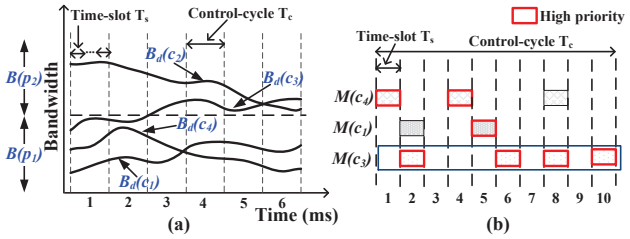

Figure 3: Illustration of LLC MPKI memory-access data-pattern: (a) bandwidth; (b) priority.

2.3 Problem Formulation

As there exists time-varying heterogeneous bandwidth demands from cores to access shared main memory, TSI I/O channels may not be fully utilized under a fixed connection. To manage 2.5D TSI I/O channels with time-varying bandwidth demands from many cores, the main idea in this paper is to learn the memory-access data-patterns so that one can perform a reconfigurable space-time multiplexing of I/Os and can improve their utilization efficiency. As such, the complexity for a large-scale demand-supply matched I/O management can be reduced with the aid of data-pattern aware smart I/Os with space-time multiplexing. One can formulate a space-time multiplexing design problem as follows:

Problem: A data-pattern aware space-time multiplexing of I/Os need to be carried out to adaptively allocate N_{ch} TSI I/Os to N_c cores such that:

$$\begin{aligned} min: \quad & \sum_{i=1}^{N_c} |B_a(c_i) - B_d(c_i)| \\ s.t.: \quad & B_a(c_i) \geq B_d(c_i); \end{aligned} \tag{1}$$

where $B_d(c_i)$ and $B_a(c_i)$ are the demanded and allocated bandwidths for core c_i respectively. The number of memory-access requests are for one control-cycle, which will be defined in the later part of this paper. In addition to the allocated bandwidth, the priority of requests also needs to be satisfied. In the following, we present a solution by studying the memory-access data-patterns.

3. MEMORY-ACCESS DATA-PATTERN

In this section, we present memory-access data-pattern analysis that will be utilised for space-time multiplexing and the QoS metric utilized for the performance evaluation. We define control-cycle with period of T_c and each control-cycle is further divided into time-slots with period of T_s. Within each control-cycle T_c, I/O channels are allocated; while at each time-slot T_s, cores are allocated with I/O channels.

3.1 LLC MPKI Pattern

There exist many approaches to describe the memory-access data-pattern of workloads. Last-level cache (LLC) misses-per-kilo-instructions (MPKI) is an important metric to indicate the communication intensiveness between cores and memory. Higher LLC MPKI indicates larger bandwidth requirement. DRAM row buffer hit-rate is another metric to show the spatial locality of the workload. Since, this paper focuses mainly on the memory-logic communication, LLC MPKI pattern is considered for analyzing the memory-access data-pattern. Note that the memory-accesses number can be inferred by LLC MPKI [23]. Memory access number $\mathbf{M}(c_i)$ for core c_i within control-cycle T_c is the sum of access number at each time-slot T_s of T_c, given as

$$\mathbf{M}(c_i)|_{T_c} = \sum_{t=1}^{T_c/T_s} \mathbf{M}(c_i)|_t. \tag{2}$$

The bandwidth demand $B_d(c_i)$ of core c_i is related to the memory-access number $\mathbf{M}(c_i)$ by

$$B_d(c_i) = \frac{\mathbf{M}(c_i) * L_c}{T_c} \tag{3}$$

where L_c is the last-level cache line size.

Based on the memory-access number, one can classify the communication traffic pattern of cores indicated by the similar demanded bandwidth. Cores with similar demanded bandwidths can vary in arrival time of memory-access requests. As such, one can prioritize the cores based on request arrival time to allocate the channel. Priority R_i will be raised when the core c_i has early arrival request or more number of requests, in case of multiple requests arriving at same time. Core assigned with highest priority will be allocated with the 2.5D I/O channel for accessing memory.

Figure 3(a) illustrates the bandwidth demand $B_d(c_i)$ for 4 cores. At control-cycle 1, due to the bandwidth demands, cores c_1, c_3 and c_4 are allocated with bandwidth $B(p_1)$, while core c_2 with bandwidth $B(p_2)$. As such, cores with different workloads can be classified based on demanded bandwidth magnitudes, for space multiplexing.

Further within the cluster consisting of cores c_1, c_3 and c_4, all cores will compete for channel at time-slot T_s. To avoid overloading of I/O channels, a time multiplexing based on priority can be implemented. Figure 3(b) shows the allocation of channels based on the priority within the time-slot. Memory-access request from a core is shown by a rectangular box. At time-slot 1, since there is only one request from c_4, I/O channel will be allocated to it. Here, when multiple requests arrive at the same time, priority is decided based on the total number of memory-access requests, which is 2 for c_1 and 4 for c_3 at time-slot 2. So, c_3 is assigned with a higher priority and occupies the I/O channel. Core assigned with high priority is shown with red outline. Thus, the cores are further classified based on the phase or priority in time, for time multiplexing.

3.2 Quality of Service

The system performance is sensitive to long-latency memory requests because instructions dependent on the long latency load cannot proceed until the load completes. More number of memory-access served indicates a better performance. Hence, the memory controller must balance the accesses from different cores with good QoS maintenance mechanisms [10].

We evaluate the performance of the system based on the number of memory-access requests processed and hence define QoS mathematically as

$$QoS = \frac{\sum_{i=1}^{N_c} r_i \mathbf{M}(c_i)}{\sum_{i=1}^{N_c} \mathbf{M}(c_i)} \tag{4}$$

where r_i is the processed ratio of requests for core c_i. Higher QoS value indicates a better performance.

4. DATA-PATTERN AWARE SPACE-TIME MULTIPLEXING OF I/OS

To solve the demand-supply matching problem by space-time multiplexing, time varying memory-access data-pattern (LLC MPKI) of cores is first extracted and then utilized in space multiplexing for channel allocation as well as in time multiplexing for time-slot allocation.

4.1 Space Multiplexing: Channel Allocation

We first discuss the adaptive allocation of I/Os to the cores. The demand here is the bandwidth requirement from cores running workloads and the supply is the I/Os to support the required access to memory. To deal with large number of cores, we cluster the cores based on the memory-access data-pattern (LLC MPKI).

Clustering can be defined as the process of grouping cores with similar demanded signature bandwidth. Each cluster is allocated with cores and connected to one of the memory controller ports through on-chip routers. Inside the memory controller, cores are virtually partitioned and allocated to a port of the reconfigurable memory controller which can meet the demanded bandwidth. This clustering of cores based on the memory-access bandwidth helps in improving the performance of low memory-intensive applications by mitigating the stalling of memory-access requests from low memory-intensive applications without affecting the performance of memory-intensive applications [17]. The number of cores for different clusters may be different. Note that the configuration of virtual clusters changes adaptively. If the demand does not vary, the clustering is expected not to change.

For example, z-th cluster G_z at port p_z is formed by

$$G_z = \{c_i | B_d(c_i) \leq B(p_z); c_i \in C, z = 1, 2, ..., M\} \quad (5)$$

where $B_d(c_i)$ is the bandwidth demanded from core c_i; and $B(p_z)$ is the bandwidth allocated to port p_z of the memory controller. Such a clustering by the magnitude of the memory-access is called space multiplexing. However, this space multiplexing overloads a port, hence a time multiplexing is needed to avoid this overloading.

4.2 Time Multiplexing: Time-slot Allocation

The time-slot allocation for time multiplexing can be described as follows. Memory-access requests from different cores can arrive at different time-instants and each can have different memory-access number. The core with earliest request will access the I/O channel first.

Memory access request from a core with early arrival time will be assigned with high priority, defined as

$$R_i = H \ if \ t_a(c_i) < t_a(c_j). \quad (6)$$

where H indicates high priority and $t_a(c_i)$ indicates the arrival time of the memory-access request from core c_i; and R_i indicates the priority for core c_i. In case, when multiple requests arrive at same time, the priority is assigned to the core with more number of memory-access requests. This priority based time multiplexing mitigates the overloading caused by space multiplexing.

It needs to be noted that the priority can change due to the change in the memory-access data-pattern (LLC MPKI), and so does the multiplexing of I/Os.

4.3 Space-Time Multiplexing of I/O

The space-time multiplexing based I/O management inside the memory controller is given in Algorithm 1.

Algorithm 1 Proposed I/O Management

Input: Set of cores C, ports P, bandwidth demands $B_d(c_i)$
1: **for** $i = 1 : N_c$ **do**
2: G_z=$\{c_i | B(p_{z-1}) < B_d(c_i) \leq B(p_z);\ z = 1, 2, ..., M\}$
3: **end for**
4: channel allocation for each cluster
5: **for** $z = 1 : M$ **do**
6: With all cores inside G_z
7: **for** $t = 1 : T_c/T_s$ **do**
8: **if** $t_a(c_i)$=min$(t_a|_{G_z})$ **then**
9: $R_i = H$
10: **else if** $t_a(c_i) = t_a(c_j)$ **then**
11: $R_i = H$ if $\mathbf{M}(c_i) > \mathbf{M}(c_j)$
12: **end if**
13: **end for**
14: **end for**
Output: cluster G, allocated I/O channels

Initially, set of ports P are labeled in ascending order of bandwidth it can provide i.e., $B(p_z) > B(p_{z-1})$. Cores with similar bandwidth demands are grouped into one cluster and connected to one port with required TSI I/Os. As such, $B(p_z) = B_a(c_i) > B_d(c_i)$. This is presented in Line 1-4 of Algorithm 1. Once the cores are clustered, priority based scheduling will be performed at each time-slot. The requests from cores will be processed on a first-come-first-serve principle. At a time-slot, if there is only one request the corresponding core will be served first (Line 8-9); If there are more than one requests, then the core with more number of requests is assigned higher priority and served (Line 11). Here 'served' indicates the TSI I/O channel will be occupied by the core in need. This process is repeated for every control-cycle T_c. Thus, I/O channel management can be performed in a space-time multiplexed manner to improve I/O utilization and QoS. Due to demand-supply matching, the utilization rate of TSI I/Os are improved. This improvement of QoS and improved I/O utilization rate helps to achieve a better energy-efficiency. The overhead of the proposed work can be the extra control logic required.

5. SIMULATION RESULTS

5.1 System Setup

In order to validate the data-pattern aware memory controller for the 2.5D TSI I/O management, system-level cycle-accurate simulation is performed. Gem5 simulator [24] is utilized for many-core microprocessors and DrSim simulator [25] is employed for shared main memory. The proposed space-time multiplexing 2.5D I/O management using reconfigurable memory controller is implemented inside the DrSim simulator. Table 1 summarizes the system design specifications. The TSI I/O channel model is based on [7] and the length is $1.5mm$. The crossbar switch-network inside memory controller is estimated with $1mm^2$ area and 1GHz frequency for a 64-core system under $32nm$ design [22]. The benchmarks are selected from SPEC 2006 [19] with high memory-access demand, PARSEC [20] with medium memory-access demand and Phoenix [21] with less memory-access demand. We set the control-cycle T_c and time-slot T_s as $1ms$ and $0.1ms$ respectively based

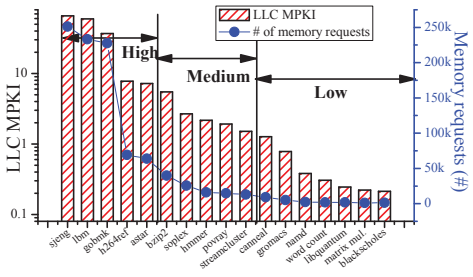

Figure 4: Analysis of memory-access data pattern.

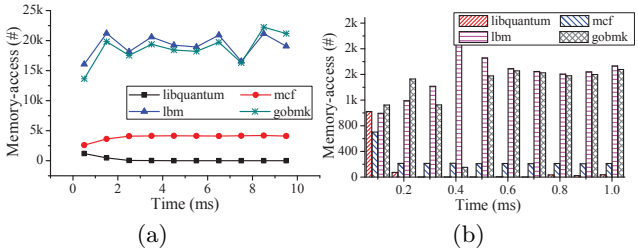

Figure 5: SPEC 2006 memory-access data-patterns: (a) for execution time of $10ms$; (b) for one control-cycle.

on switching speed. Furthermore, we assume a baseline system which has fixed core-to-memory connections, while the proposed system employs the reconfigurable connections. The overhead in terms of hardware for the data-pattern aware smart I/O is the additional area required for crossbar switch. The latency caused by crossbar is small compared to the chosen control-cycle.

Table 1: System parameters

Processor core	1GHz x86
L1 I-cache	32kB private, 64B cache line
L1 D-cache	32kB private, 64B cache line
L2 cache	256kB private, 64B cache line
Main memory	4GB capacity, 800MHz DDR3-1066 channel, x8 DRAM chips, 8 banks per channel

5.2 Memory-access Data-pattern Analysis

Memory-access data-patterns can be classified based on the bandwidth demands (LLC and number of memory requests) is presented in Figure 4. We classify benchmarks as high, medium and low memory-access benchmarks. For example, a core running *lbm* benchmark having LLC MPKI of 59 (represented by vertical bar) and 233K memory requests can be categorized under high memory-access, whereas core running *libquantum* is classified under low memory-access benchmarks due to low LLC MPKI of 0.24 and 1776 memory requests. One needs to note that LLC and number of memory-access requests are proportional.

Further, we present how the data-patterns with similar memory-access numbers can be classified. From Figure 5(a), one can observe that *lbm* and *gobmk* has similar bandwidth signature and high memory-access demands compared to *libquantum* and *mcf*. Thus, *lbm* and *gobmk* can form a cluster (space multiplexing). In Figure 5(b) we present the memory-access number within one control-cycle. Variation in number of memory-access demands of cores in one cluster (*lbm* and *gobmk*) can be observed, based on which priorities can be assigned to occupying I/O channels in time multiplexed manner.

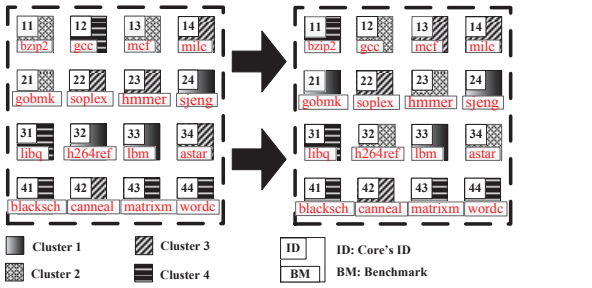

Figure 6: Adaptive clustering of cores at two consecutive control-cycles.

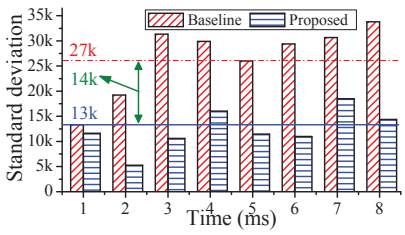

Figure 7: Bandwidth balance across TSI I/O channels of 64-core under randomly distributed 16 benchmarks of SPEC 2006, Parsec and Phoenix.

5.3 Adaptive Clustering Analysis

Here, we present the adaptive clustering of cores based on the signature of demanded bandwidth. For a 16-core case, we assume the total number of I/O channels are 8 and the number of ports is 4 and each core is randomly assigned with a benchmark. Hence, for the baseline system with fixed connections, each port is assigned with four cores and each port is connected to 2 I/O channels. While for the proposed system, ports connect to 4, 2, 1 and 1 I/O channels respectively to meet the memory-access demands from high to low traffic benchmarks. A similar setup is assumed for 64-core case. Figure 6 illustrates the adaptive clustering result of 16-core case at two consecutive control-cycles ($5ms$ to $6ms$). Different filling patterns represent different clusters. Cluster 1 handles high-traffic workloads, cluster 2 are used for middle-traffic workloads, and cluster 3 and cluster 4 are allocated with low-traffic workloads. For example at $5ms$, core 12 running *gcc* benchmark is assigned to low traffic cluster 4, but is assigned to middle traffic cluster 2 in the next control-cycle due to time varying memory-access characteristics.

Bandwidth balancing across all ports can improve the I/O channel utilization efficiency. We use requests per channel to measure the traffic flow at each control-cycle and calculate the standard deviation for all four ports. The standard deviation shows the variation from average value. A low standard deviation indicates more bandwidth balancing. Bandwidth balancing for a 64-core processor, with 52 SPEC 2006, 8 Parsec and 4 Phoenix benchmarks randomly allocated to cores is shown in Figure 7. For example, at $5ms$ the baseline system show standard deviation of 26K memory requests, while the proposed system just requires 11.5K with 55.90% bandwidth balancing improvement. The average of standard deviation for baseline and proposed system is 27K and 13K memory requests, indicating a 14K reduction in deviation. On average the proposed system can improve the bandwidth balancing by 58.85% under 64-core case.

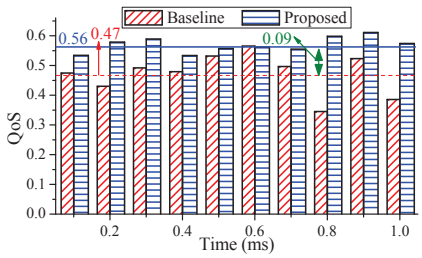

Figure 8: Communication QoS efficiency improvement by STM I/O management for 16-core.

5.4 QoS Analysis

For a 16-core microprocessor with SPEC 2006, Parsec and Phoenix benchmarks randomly distributed on cores, comparison of QoS for proposed system and baseline architecture is presented here. Improvement in QoS by the proposed system is shown in Figure 8. The average QoS for the baseline system is 0.472, whereas for the proposed space-time multiplexing achieves a QoS of 0.561. Further considering a $10ms$ span as an example, the average QoS achieved by the proposed, baseline and time multiplexing schemes are presented in Table 2. For a 16-core case, a QoS of 0.589, 0.473 and 0.528 are achieved by proposed, baseline and time multiplexing schemes, respectively. This indicates nearly 11.90% more requests are served and improvement in QoS by the proposed method compared to baseline. Whereas for 64-core case the proposed system achieves a QoS of 0.577 compared to QoS of 0.461 and 0.514 achieved by baseline and time multiplexing techniques respectively. Use of space multiplexing alone may result in large congestion at memory controller, hence not compared. As mentioned in the previous section, the improvement is achieved from the performance improvement of low memory-intensive applications (space multiplexing) and mitigation of request stalling (time multiplexing).

Table 2: QoS Comparison

For 16-core		
Method	# Requests served	QoS
Baseline	448578	0.473
Time multiplexing	523840	0.528
Proposed	**580399**	**0.589**
For 64-core		
Method	# Requests served	QoS
Baseline	2092083	0.461
Time multiplexing	2340843	0.514
Proposed	**262005**	**0.577**

6. CONCLUSION

In this paper a smart I/O, which is a data-pattern aware I/O with space-time multiplexing is demonstrated for 2.5D memory-logic integration. With the reconfigurable crossbar switch-network inside the memory controller, bandwidth demand from many-core to access the shared memory can be managed when accessing with limited 2.5D TSI I/O channels. A space-time multiplexing based communication between cores and memory is realized by reusing the I/O channels with improved communication efficiency. By adaptive clustering of the cores upon the magnitude of memory-access patterns, 2.5D I/O channels are allocated to core clusters by space multiplexing. With further considering priority upon the phase of memory-access patterns, time-slots are allocated to access 2.5D I/O channels in one cluster by time multiplexing. The proposed architecture is verified by the system-level simulator with benchmarked workloads, which shows up to 58.85% bandwidth balancing and 11.90% QoS improvement.

7. REFERENCES

[1] A. Vahidsafa and et.al., "SPARC M6: Oracle's next generation processor for enterprise systems," in *HOT CHIPS*, 2013.

[2] M. B. Healy and et.al., "Design and analysis of 3D-MAPS: a many-core 3D processor with stacked memory," in *IEEE CICC*, 2010.

[3] M. P. D. Sai and et.al., "Reliable 3-D clock-tree synthesis considering nonlinear capacitive TSV model with electrical–thermal–mechanical coupling," *IEEE Trans. on CAD*, vol. 32, no. 11, pp. 1734–1747, Nov 2013.

[4] M. P. D. Sai, H. Yu, and K. Wang, "3D many-core microprocessor power management by space-time multiplexing based demand-supply matching," *IEEE Trans. on Computers*, vol. PP, 2015.

[5] "Hybrid Memory Cube Consortium," http://hybridmemorycube.org/tool-resources.html.

[6] H. Yu, J. Ho, and L. He, "Allocating power ground vias in 3D ICs for simultaneous power and thermal integrity," *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, vol. 14, no. 3, p. 41, 2009.

[7] J. R. Cubillo and et.al., "Interconnect design and analysis for through silicon interposers (TSIs)," in *IEEE 3DIC*, 2012.

[8] A. Vassighi and et.al., "Thermal runaway in integrated circuits," *IEEE Tran. on DMR*, vol. 6, no. 2, pp. 300–305, Jun 2006.

[9] S.-S. Wu and et.al., "A thermal resilient integration of many-core microprocessors and main memory by 2.5D TSI I/Os," in *ACM/IEEE DATE Conf.*, 2014.

[10] M. K. Jeong and et.al., "A QoS-aware memory controller for dynamically balancing GPU and CPU bandwidth use in an MPSoC," in *ACM/IEEE DAC*, 2012.

[11] B. Akesson and et.al., "Memory controllers for high-performance and real-time MPSoCs requirements, architectures, and future trends," in *ACM/IEEE Int. Conf. on Hardware/Software Codesign and System Synthesis*, 2011.

[12] Denali Software Inc., "Databahn DRAM memory controller IP," 2009.

[13] S. Rixner and et.al., "Memory access scheduling," in *Int. Symp. on Computer Architecture*, 2000.

[14] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems," in *Int. Symp. on Computer Architecture*, 2008.

[15] E. Ebrahimi and et.al., "Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems," in *ACM Architectural Support for Programming Languages and Operating Systems*, 2010.

[16] Y. Kim and et.al., "Thread cluster memory scheduling: Exploiting differences in memory access behavior," in *IEEE/ACM Int. Symp. on Microarchitecture*, 2010.

[17] S. P. Muralidhara and et.al., "Reducing memory interference in multicore systems via application-aware memory channel partitioning," in *IEEE/ACM Int. Symp. on Microarchitecture*, 2011.

[18] R. Das and et.al., "Application-to-core mapping policies to reduce memory system interference in multi-core systems," in *IEEE Int. Symp. on High Performance Computer Architecture*, 2013.

[19] "SPEC CPU2006 Benchmark," http://www.spec.org/cpu2006/.

[20] "PARSEC Benchmark," http://parsec.cs.princeton.edu/.

[21] C. Ranger and et.al., "Evaluating mapreduce for multi-core and multiprocessor systems," in *IEEE Int. Symp. on HPCA*, 2007.

[22] K. Sewell and et.al., "Swizzle-switch networks for many-core systems," *IEEE J. on Emerging and Selected Topics in Circuits and Systems*, vol. 2, no. 2, pp. 278–294, Jun 2012.

[23] A. Hilton, S. Nagarakatte, and A. Roth, "iCFP: Tolerating all-level cache misses in in-order processors," in *IEEE Int. Symp. on HPCA*, 2009.

[24] N. Binkert and et.al., "The gem5 simulator," *ACM SIGARCH Computer Arch. News*, vol. 39, no. 2, pp. 1–7, 2011.

[25] M. K. Jeong and et.al., "DrSim: A platform for flexible DRAM system research," http://lph.ece.utexas.edu/public/DrSim.