NORTH CAROLINA STATE UNIVERSITY

Analyzing the Effect Of Predictability of Memory References on WCET

Amir Bahmani
Department of Computer Science
North Carolina State University
abahaman@ncsu. edu

Vishwanathan Chandru
Department of Computer Science
North Carolina State University
vchandr6@ncsu.edu

Guided By
Dr. Frank Mueller
Department Of Computer Science
North Carolina State University

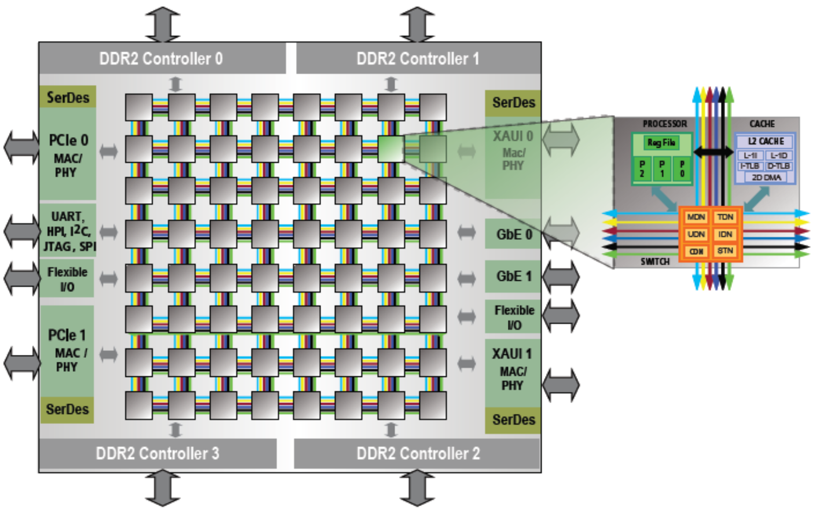
Abstract

In a commercial multi-core system, multiple cores share many resources including DRAM. In such a multi core system, two completely unrelated tasks can cause cross interference, especially in terms of memory access. This variable and unpredictable delay poses a significant challenge when it comes to predictability and isolation. Given the weighing of predictability over performance in real time systems, it is a highly undesirable characteristic. In this project we target a particular NoC (Network on Chip) architecture for analysing memory contention and unpredictability. TileraPro64 has 64 core connected by a mesh interconnect which is used for communication between tiles and between memory controllers. Platforms consisting of several cores (multi-cores) and more than a dozen of cores (many-cores) have nowadays become the mainstream in many scientific areas, most notably high performance computing, while are the new frontier technology in others like real-time embedded systems. Along with positives of whole lot of processing power and increased system availability, this comes with multiple latencies, especially in terms of memory accesses as multiple cores try to access the memory at the same time. This becomes even worse with a multi-processor board like Tilera which is having processors in order of 50's and above, NoC added to that unpredictability. As a part of this project we propose a controller aware memory controller which reduces the serialization of memory accesses resulting into a better predictability and performance isolation. As a part of this project we also try to quantify the impact of memory bound tasks on WCET.

Introduction

When it comes to multi core systems, all cores share the same memory. Thus inefficient use of memory can easily become a performance bottleneck and a source of unpredictable behaviour. A short literature survey to understand the delay or unpredictability added due to contention of resources especially memory. First was the contention to access the same memory when memory is not divided into banks or bank aware allocation is not performed. Two analysis procedures found in this context request driven analysis and job driven analysis. It was also found about various inter-bank and intra bank interferences result to unpredictability [1]. We found one of the bank aware allocator called PALLOC which alleviated this problem to a certain extent [2]. But static/dynamic partitioning alone cannot solve this problem as when we have a board like Tilera with large number of cores and NoC packets being sent across for memory request and communication predicting becomes tricky [3]. We need to consider worst case latency in this scenario using per pattern analysis. Also if we consider involvement of multiple memory controllers and live process migration for load balancing situation becomes even trickier. So we concluded that we need a multi faced solution which involves distributing memory requests across controllers to reduce latency, static/dynamic banking of memory and optimal placement of processes among cores to ensure minimum n/w traffic and minimum latency, thus a more predictable and reliable WCET. So we concluded that we need a multi faced solution which involves distributing memory requests across controllers to reduce latency, static/dynamic banking of memory and optimal placement of processes among cores to ensure minimum n/w traffic and minimum latency, thus a more predictable and reliable WCET.

In this project we target TileraPro64 hardware platform. This board features 64 identical cores (alternatively called tiles), each connected via mesh interconnect. It features various networks for communication namely IDN (I/O dynamic network for OS usage and streaming data), MDN (Memory Dynamic Network for loads/stores/pre-fetches/cache misses/DMA), UDN (User Dynamic Network, used in BME), CDN (Coherence Dynamic Network for L3 invalidations), TDN (Tile Dynamic Network for used by cache for core-to-core block transfers). Each tile/core is fully fledged processor consisting of L1 and L2 cache. It features a soft L3 cache composed by sharing L2 caches of all processors. It features 4 memory controllers each controlling upto 16 GB of memory. It has 32 bit virtual address space and a 36 bit global physical address space.



In this project we try to analyze and quantify the impact of memory contention on WCET and isolation of tasks in terms of memory. There are two possible approaches to be considered. First approach is a kernel level memory aware allocator and alternative approach is user space memory allocator. This is explained in detail in next section.

Proposed Model

As mentioned before, as a part of this project we are targeting only distribution of memory accesses across controllers so as to reduce serialization and ensure a more predictable WCET and task isolation, there are two methodologies for this purpose. First one is kernel level memory aware allocator and second one is user space allocator. Key to both the approaches is in figuring out how virtual address is translated into physical address and then identifying the how a physical address is mapped onto a respective memory controllers. Investigation was performed for the feasibility of first methodology and started off with the prospect of intercepting the L2 misses. The structures governing client configurations were identified namely client_config, but could not locate any library static or dynamic which exports the symbols for the necessary functionality. The already exported symbols available were analyzed and were not of any use. It was concluded that to carry on in this direction it would require a hypervisor rebuild and deployment. Another perspective to approach was to do the loading of pages into memory adhering to the stripping and controller assignment done at virtual address level instead of remapping on the fly. It was found that hypervisor is called only when needed and one particular need which could solve problem of mapping is page fault handling. It does something called downcall, which is basically telling the underlying OS to act if we need kernel to act. In our context, if the page couldn't be located on L1 or L2 cache(s) than control is transferred to OS to handle the PF. We can take advantage of this downcall to handle the PF the way we want and this allows us to control both user and kernel space allocations. This approach is similar to the approach taken in "PALLOC: bank aware memory allocator". Also as part of the investigation, it was found that the 36 bit address used in MDN packets can be used to identify the bank, row, MC Etc. Three types of addresses were identified 32 bit VA, 64 bit client physical address as seen by hypervisor, 64 bit PA as seen by MC. Although 64 bit PA is supported but only 36 bit global shared PA are used as of now. Downcall handling is best way to solve the remapping but it requires hypervisor modification and along with that some page fault handling modification which is not feasible as of now.

Page table functions are identified and corresponding interrupt handlers are located but the understanding of code and the exact control flow once the control enters the Linux kernel so as to modify allocations is not clear. Also it is not clear about how to build and deploy the modifications. Interestingly these involve downcall modifications. Given that and source code for hypervisor we could define a design and implementation for coloured malloc. Due to these reasons, the former and better approach was deferred. Investigation for the user space controller aware allocator was performed. First challenge was to figure out for this approach to work out was to figure out mapping of addresses (Virtual Address/Physical Address) to controller. It was found in the documentation that out of 36 bit physical address, Two most significant bits are used to identify MC and rest 34 for identifying address. So in a way each memory controller can address a max of 16 GB of memory. The conversion of 36 bit to external address depends on whether or not hashing is enabled. The external address if of following format:

Chip (rank)	Row	Bank	Column

Figure 2-2: External Memory Address Format

The bits for row, bank and column depends on how memory is configured and distributed among memory controllers.

.

Table 2-80. Supported DDR2 Devices

Configuration	Bank Bits	Row Bits	Column Bits
256Mb x8	2	13	10
256Mb x16	2	13	9
512Mb x8	2	14	10
512Mb x16	2	13	10
1Gb x8	3	14	10
1Gb x16	3	13	10

For our case, the memory configuration is 2 GB per controller, so configuration corresponds to first entry the table i.e. the external address in our case should be 25 bits plus 2 bits for chip (rank), thus 27 bits are used for external address. The hypervisor relies on downcall mechanism for page fault handling which translates into fact that Linux kernel handles the page fault for hypervisor. It means that ***/proc/pid/pagemap*** can be utilised to find out the physical frame number can be used to convert the virtual address to physical address. One more factor to be considered in our scenario is memory striping (enabled by default). What it does is distribute the page among controllers with granularity of 8 kb. So if we take a page of size 64k, the first 8k goes to MC 0, next 8k to MC 1, next 8k to MC 2 and so on. This is enabled by default in the Tilera board if each controller has equal memory and as long as this option is enabled Linux sees only one unified memory controller handling entire memory. After further reading through the architecture document it was figured out that bits determining the memory controller vary based on what mode are we running on. If memory striping is enabled, bits 13 and 14 of 36 bit PA determine the memory controller else two most significant bits determine the controller. The 36 bit address is generated by composing higher bits using page frame number and lower bits using the page offset taken from virtual address. In this approach we allocate a huge chunk of memory creating a pool of memory. Then we touch the pages to load them into physical memory and identify the controllers the page is mapped to. After identifying the mapping assign a colour to each page depending on the granularity of page size (in case of striped memory turned off) and 4k in case of memory striping turned on. Proposed allocator allocates in multiples of granularity of allocation. When it comes to allocation requests there are three possible approaches:

1) Whenever memory is requested from pool, generate a random colour, and try to allocate a memory of generated colour. If not possible, allocate any free chunk. This approach is bit non uniform and could lead to exhaustion of a particular colour.

2) This approach requires caching of the last allocated colour. Whenever an allocation is requested, we increment the cached colour by 1 and search for the memory of that particular colour. If found well and good, but if not found allot whatever free slot was found and update the cached colour with the colour of allocated memory.

3) User requests a particular colour and allocation of that particular colour is performed.

These approaches work fine only if requested allocation is less than or equal to granularity. If not we find the contiguous chunk of memory satisfying size constraints and return that. These approaches should be fine with both striped memory and inter leaved page allocation. The granularity of allocation is 8k when in striped memory mode and page size when in inter leaved page allocation mode. Current implementation supports approaches 2 and 3.

Implementation

First step towards the implementation is extraction of bits determining the memory controller. Experimentation was first performed on 26th and 27th bits of final physical address. But it was found that behaviour was not what was expected, in many cases we get an address greater than 27 bit and chip (rank) bits are no more the most significant bits as they should be ideally. Also if we try to extract the bits 26 and 27 the controller id is not what we expect it to be as per the striping behaviour mentioned earlier. For e.g. in one case all the 8k sections within a page have 0 in their 26th and 27th bit which is not true. After further investigation and reading the various architectural documentations again, the understanding of memory address translation, testing was done with extracting the 13th and 14th bit from the page offset. It turned out that we indeed get a different memory controller every 4k (2^12) and given a page size of 64k it is distributed across memory controllers with a granularity of 8k. Next attempt was made to disable memory striping and do inter leaved page allocation. Goal was to extract 34th and 35th bit to verify that those bits indeed determine the controller. We tried disabling the stripe memory mode using the hypervisor configuration file but some internal file kept enabling the mode due to which verification for controller determination at page level could not be performed. To reduce the variability due to page replacement, tmc_alloc_map is used. It has three advantages. First one is, this api allows to disable the L1 and L2 cache (hashing and DSM in turn). Second one being the functionality of tmc_alloc_map being similar to mmap. Third one being that it does not do lazy fetching, i.e. the pages for the requested memory are immediately allocated in main memory and it does not wait for pages to be touched. This comes as a great advantage when are trying to implement a coloured malloc in the user space.

Experiment And Results

As part of contention analysis, impact of NoC contention is measured using repeated memory accesses considering only RAW, WAR and WAW dependencies. To reduce the unpredictability, L1 and L2 cache(s) are disabled. Disabling the caches ensures hashing and homing of data which in turn ensure every memory access request goes to main memory even if L1/L2 cache have the required data. The plan is to use pthreads to emulate tasks. Since we have 4 memory controllers we have 4 threads. Chunks of size 8k and 4k are allocated with the use of our allocator and without the use of proposed allocator. Chunks are accessed consecutively and writes are performed. Each task accesses a fixed number of chunks and all chunks are accessed for a fixed number of times in a cyclic manner. This ensures that even if caching is enabled every memory access will be a cache miss. Memory latency is measured in terms of cycles and various placements of threads are tried.

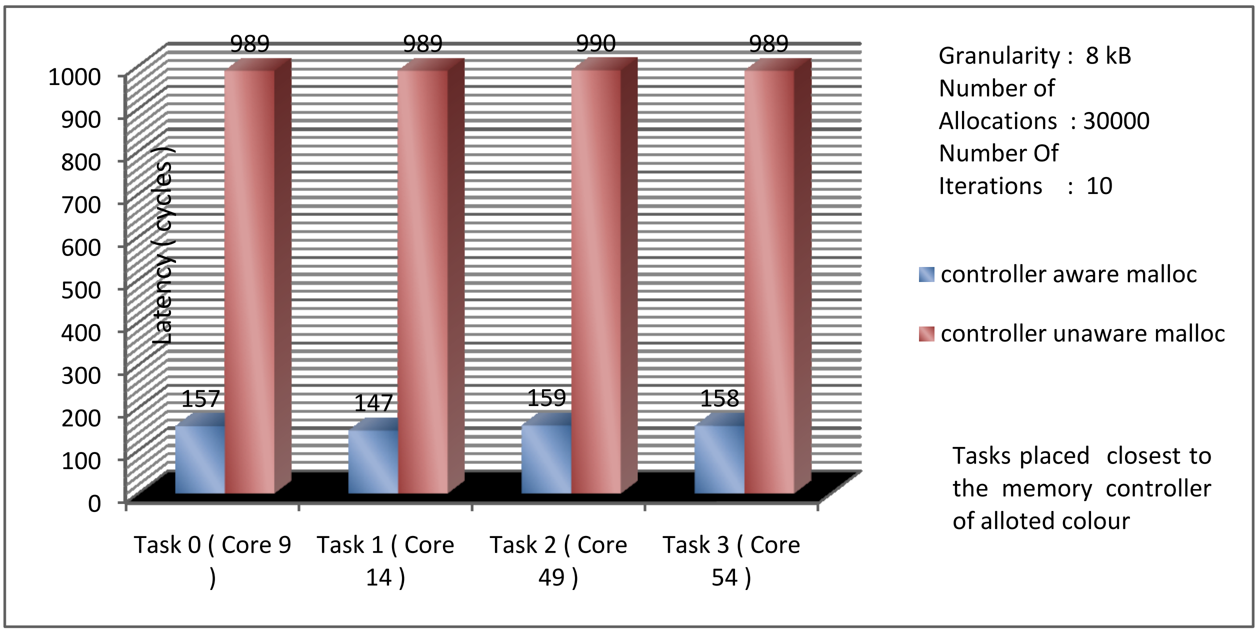


Fig 1

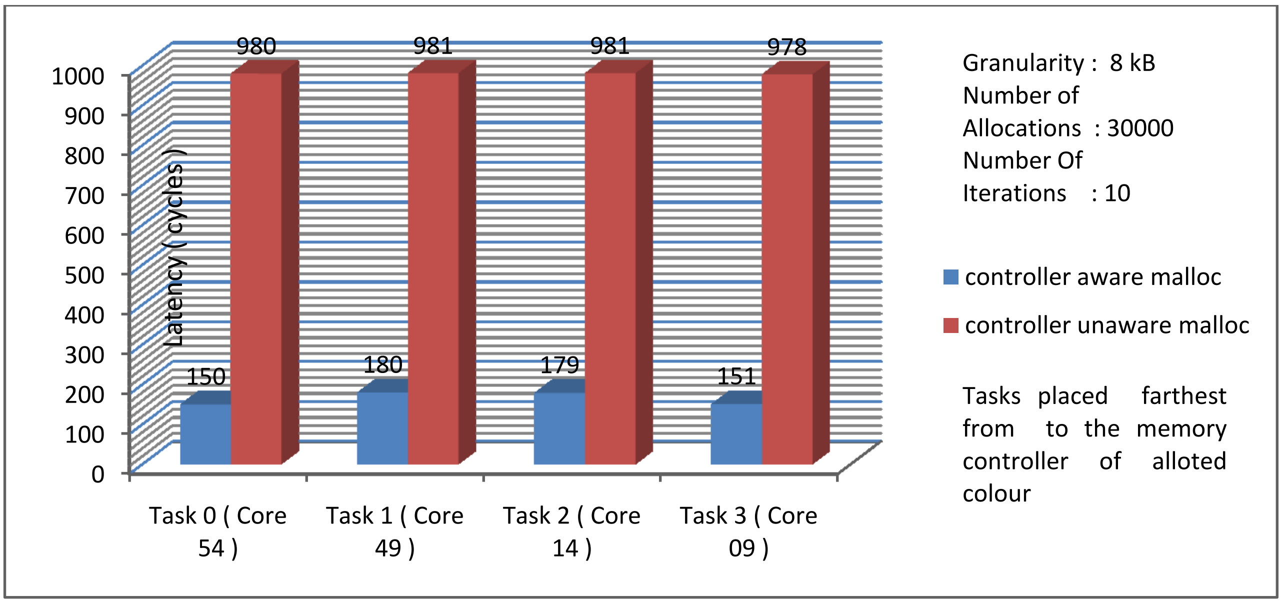


Fig 2

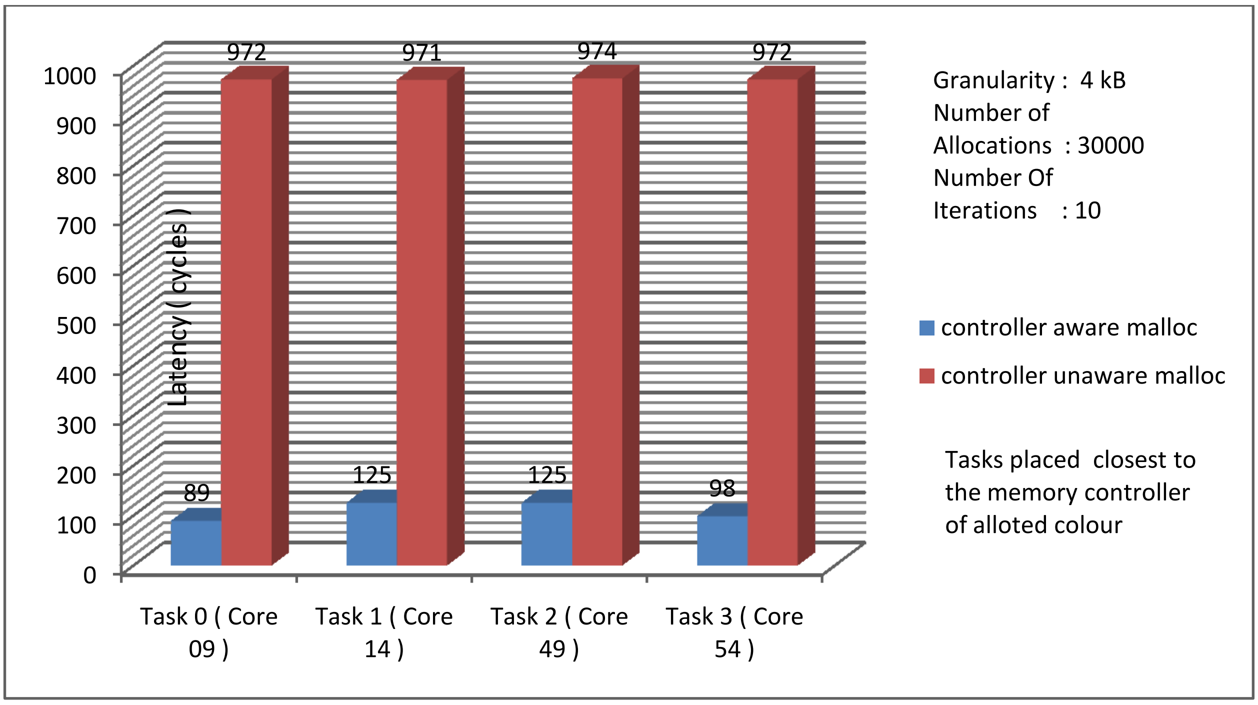


Fig 3

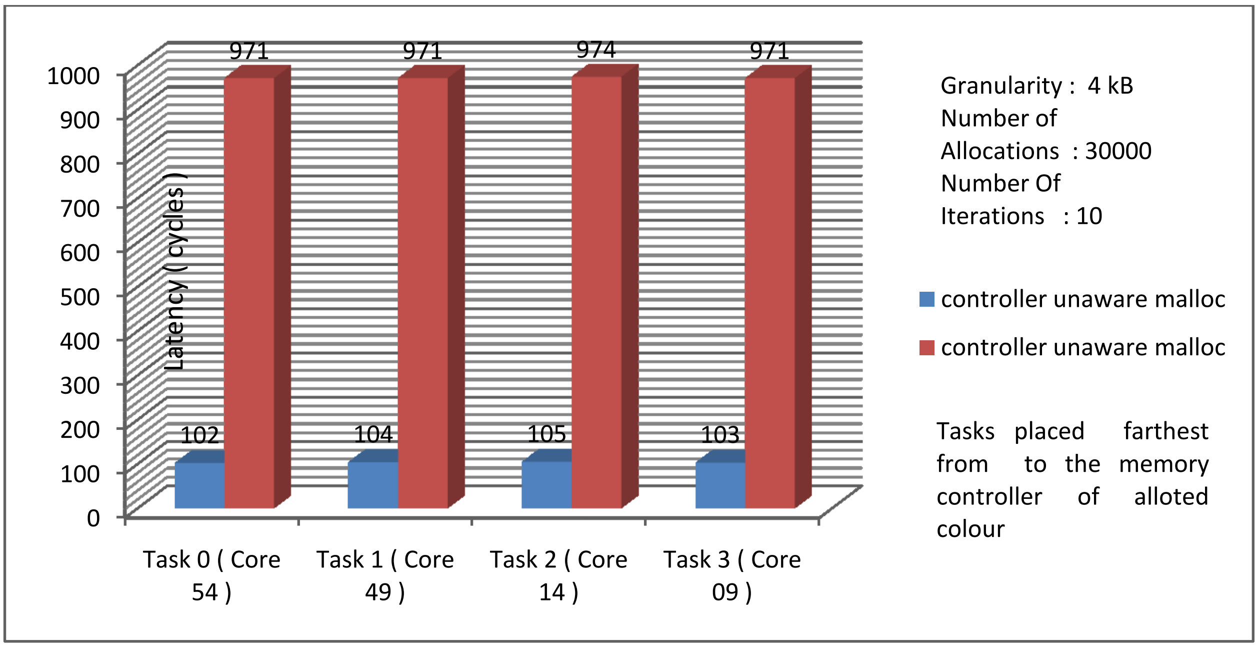


Fig 4

As we can see from the plots, if we follow a 4 kB allocation granularity we get the least latency and best possible isolation as stripe size is 4 kB (13th and 14th bits are used to determine allocation which means each controller gets two chunks of size 4 kB). Also since our experiment was very much limited to only 4 task set and no interfering tasks, we do not see any big impact of task placement on memory access latency.

Conclusion

As shown by the results, the proposed controller aware allocator can reduce memory latency and is capable of reducing the unpredictability leading to better isolation and tighter WCET bounds. Experiments could only be performed for memory stripe mode. Although the allocator will work for non stripe mode allocations by design but stripe mode needs to be disabled and allocator needs to be tested on interleaved page allocation.

Future Work

This project only covers a user space allocator which inherently restricted and sub optimal by design. We need to extend the concept of coloured malloc and make it a kernel level entity working at a level of page fault handling. As of now if we want to control the allocation restricted to a memory controller, API support is there but will involve a lot of work from user and thus user may not always do an optimal implementation. A new abstraction above the existing API and modifying PF handling routine to allocate physical page frames considering parameters like available MC bandwidth, NoC contention, and distance from memory controllers can further boost predictability especially when it comes to memory bound applications. One more thing which needs to be investigated is impact of task migration on memory allocations. Live migration is very important when it comes to power consumption and predictability and needs to be considered while doing controller aware memory allocation. In our experiments, tasks were pinned to the core/tile and hence results do not indicate the effect of migration.

Task Status

Task	Status	Open Actions
Ramp up on Tilera architecture and APIs	Closed (24th March 2014) Owner : Both	
Figuring out base parameters for coloured malloc	Closed (24th March 2014) Owner : Vishwanathan	
Figure out way to remap the memory accesses	Deferred Owner : Vishwanathan	
Figure out a way to figure out the corresponding MC from physical address	Closed (14th April 2014) Owner : Vishwanathan	
Memory mapping strategy	Closed (14th April 2014) Owner : Vishwanathan	
Strategy for contention analysis	Closed (14th April 2014) Owner : Vishwanathan	
Malloc implementation	Closed (14th April 2014) Owner : Vishwanathan	
Test Cases Design and Implementation	Closed (14th April 2014) Owner : Vishwanathan	
Task Mapping Algorithm and implementation	Deferred Owner : Vishwanathan	
Task Mapping algorithm validation	Deferred Owner : Vishwanathan	
Presentation	Deferred Owner : Vishwanathan	
Final Report	Closed (1st May 2014) Owner : Vishwanathan	

Source Code

Following github url can be used to clone the repository

git@github.ncsu.edu:vchandr6/rtcs.git

Please send me a mail with ssh public key so that I can add you to the repository.

References

[1] Hyoseung Kim , Dionisio de Niz, Bj orn Andersson† , Mark Klein†, Onur Mutlu, Ragunathan (Raj) Rajkumar , Bounding Memory Interference Delay in COTS-based Multi-Core Systems in COTS-based Multi-Core Systems

[2] Heechul Yun , Renato Mancuso , Zheng-Pei Wu , Rodolfo Pellizzoni, PALLOC: DRAM Bank-Aware Memory Allocator for Performance Isolation on Multicore Platforms

[3] Borislav Nikoli´ c, Patrick Meumeu Yomsi and Stefan M. Petters, Worst-Case Memory Traffic Analysis for Many-Cores using a Limited Migrative Model

[4] https://android.googlesource.com/kernel/common.git/+/android-3.0/

[5] Tilera Architecture documentation

[6] Porting Barrelfish to the Tilera TILEPro64 Architecture, ROBERT RADKIEWICZ and XIAOWEN WANG, KTH Information and Communication technology

[7] Cacheaware Parallel Programming for Manycore Processors, Ashkan Tousimojarad and Wim Vanderbauwhede, School of Computing Science, University of Glasgow, Glasgow, UK

[8] http://fivelinesofcode.blogspot.com/2014/03/how-to-translate-virtual-to-physical.html

[9] Many-Core Key-Value Store, Mateusz Berezecki, Eitan Frachtenberg, Mike Paleczny , Facebook, Kenneth Steele, Tilera

[10] TILE-Gx100 ManyCore Processor: Acceleration Interfaces and Architecture, Carl Ramey ,Principal Architect, Tilera Corp

[11] UG104-IO-Device-Guide

[12] Architectures for Multimedia Systems, TILERA – TILE64™ PROCESSOR, Mondello Filippo

[13] UG101-User-Architecture-Reference