Final Report

Xing Pan (xpan)

Project introduction:

In modern CPU, there are many cores and several memory controllers in one processor. For example, in the 2-way SMPs with AMD Opteron 6128, there are 16 cores and 4 memory controllers in 1 node. Each core access different memory controllers will get the different latency. So it is not an easy work to predict the worst case execution time when we run the program on the multicore system.

The goal of this project is to find a new virtual and physical address mapping algorithm help us to predict the memory access latency exactly in the multicore system.

Tasks accomplished:

Step 1: verify there is the memory latency existed.

To verify the memory access latency existed, I did the brute-force test program on the 2-way SMPs with AMD Opteron 6128. Use ”go back and forth” to avoid the hardware prefetch and ”calloc a large address space” to avoid the cache hit. Utilize the /proc/pid/pagemap and /proc/pid/maps file to map the virtual address to physical address. Get the following data to verify that there exist the latency which caused by accessing multiple memory controllers.

Virtual address : 0x2B82002EB010

Physical address: 0x1DAF32010,

Node: 0, channel: 1, rank: 3, bank: 2, row: 6959, column: 512

latency: 8111us *

Virtual address : 0x2B82FE2ED010

Physical address: 0x2055BE010,

Node: 0,channel:0, rank: 3, bank: 3, row: 15829, column: 2560

latency: 12466us

Virtual address : 0x2B8535AEF010

Physical address: 0x611485010,

Node:2,channel: 1, rank: 2, bank: 0, row: 16020, column: 2304

latency: 17152us

Virtual address : 0x2b85f8af0010

Physical address: 0x556E74010,

Node: 2, channel: 1, rank: 3, bank: 6, row: 4846, column: 2048

latency: 17006us

Virtual address : 0x2b86f7af1010

Physical address: 0x80ED03010,

Node: 3, channel: 1, rank: 2, bank: 0, row: 7789, column: 768

latency: 16957us

The data clearly shows that the latency from accessing different memory controller. For example, if the core access node 0 (memory controller id 0) will get the least latency. However, if it accesses the node 2 or node 3, there will be more latency.

Step 2: Configure out the AMD Opteron 6128 architecture

In the experiment, I found that the physical address is grouped by the NodeID.

For AMD Opteron 6128, there are 4 nodes (memory controllers) in one CPU. Each node (memory controller) has different physical address range.

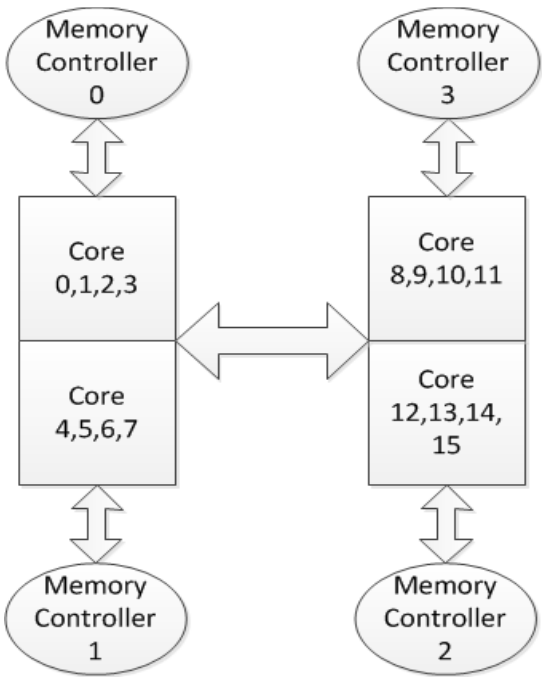
Node 0:0x0~~0x227FFFFF

Node 1:0x228000000~~0x427FFFFF

Node 2:0x428000000~~0x627FFFFF

Node 3: 0x628000000~~0x827FFFFF

There are 16 cores in one CPU. When one core in the CPU accesses the main memory, it will access different memory controller and get the different access latency. The following graph is the architecture of AMD Opteron 6128.



There are some observations:

1. When a program run on one core, the nearest memory controller is preferential.

I did another experiment about this observation. The program just calloc 8GB which is no more than the physical space in one memory controller. The result shows that the program just use the nearest memory controller.

2. It takes least latency when Core 0,1,2,3 access memory controller 0. It takes the longest latency when they access memory controller 2 and 3. However, it takes medium latency when they access memory controller 1. Same situation for the other cores. For example, for core 0

memory controller 0: 9000us *

memory controller 1: 12000us

memory controller 2,3: 17000us

Step 3 test the contend and latency between multiple threads in multicore system

Use a latency test program. In this program, it does the initialization (calloc a memory space) at this main thread and creates several threads. Each thread will bind to different core and access a same memory controller respectively.

Then I measure the accessing latency for each thread and compare the latency with the latency if there is only one thread access the memory controller (same large memory space).

The following is the data:

One thread assigned on the core 8 and access the memory controller 3 :

latency is 9400us *

Two threads, one of them assigned on the core 8 and the other one assigned to core 9. Both of them access the memory controller 3 :

the maximum latency is 12700us, the minimum latency is 10900us

Four threads, assign them to core 8, core 9, core 10 and core 11 respectively. All of them access the memory controller 3 :

the maximum latency is 19500us, the minimum latency is 12200us

Four threads, assign them to core 0, core 4, core 8 and core 12 respectively. They access the memory controller 0, 1, 2, 3 correspondingly:

latency is 9300us

There are some observations:

1. The latency is much longer than the latency if there is only one thread access one memory controller alone. (about 9000us).

2. The more threads access a same memory controller, the worse latency they get. For example, the latency when I use 4 threads is much longer than only 2 threads.

3. If we use multi-threads to access the same memory controller, there definitely is a contend. Besides, the latency is variable and irregular fluctuated. For real time system, It is very hard to predict it.

4. When a program run, it is very hard to know which memory space it uses. Although the operation system has the "first tough" mechanism, the program still use multiple memory controllers instead of single memory controller potentially.

Step 4: Coloring algorithm design and Evaluation

Design the coloring algorithm:

In this step, I designed the main memory coloring allocation algorithm based on memory controller.

The basic idea of this algorithm is group the main memory based on the memory controller. All the banks in one memory node (controller) would like to be colored in one group. For AMD Opteron 6128, there are 4 nodes (memory controllers) in one CPU and 32 GB main memory. So we color each memory node as one color. There are 4 colors in all the 32GB main memory space.

Color 0: 0x0~~0x227FFFFFF all the 32 banks in node 0, 8GB

Color 1: 0x228000000~~0x427FFFFFF all the 32 banks in node 1, 8GB

Color 2: 0x428000000~~0x627FFFFFF all the 32 banks in node 2, 8GB

Color 3: 0x628000000~~0x827FFFFFF all the 32 banks in node 3, 8GB

For a program, there are several tasks (threads) in it. This algorithm will assign the different colored main memory bank to each task. So each task will access main memory through different memory controller and use different main memory space. Use this coloring algorithm, we can predict the memory access latency exactly and get lower worst case execution time.

Evaluation:

(1) Correctness and non-contention

I did the experiment on the ARC system. There are 16 cores and 32 GB main memory space in one processor. The test program creates 4 threads and allocates 4 GB memory to each thread. Each thread accesses the main memory and the program measure the latency of it. The following data shows the memory address which accessed by each threads. I used two allocation method, the general allocation and new coloring allocation.

For general allocation, we get the data as following:

Thread 0:

VirtualAddr	PhysicalAddr	Node	Channel	Rank	Bank	Row	Col	color
0x2aac30005010	0x10005D010L	0	0	2	7	3456	2304	0
0x2aac30805010	0x104751010L	0	1	2	6	3527	256	0
0x2aac31005010	0x100091010L	0	0	2	2	11648	256	0
0x2aac31805010	0x104752010L	0	1	2	6	3527	512	0

0x2aac32005010	0x111ECF010L	0	0	2	5	11934	2816	0
0x2aac32805010	0x10D395010L	0	1	2	2	11859	2304	0
0x2aac33005010	0x109326010L	0	0	3	0	3603	2560	0
0x2aac33805010	0x10D396010L	0	1	2	2	11859	2560	0
0x2aac34005010	0x11BEB7010L	0	1	3	2	12094	2816	0
0x2aac34805010	0x116F7D010L	0	0	3	7	3823	2304	0
……								
0x2aacac805010	0x332D7E010L	1	0	3	7	4269	2560	1
0x2aacad005010	0x32F9A7010L	1	1	3	0	12409	2816	1
0x2aacad805010	0x332D7F010L	1	0	3	7	4269	2816	1
0x2aacae005010	0x336D9A010L	1	1	2	3	12525	512	1
0x2aacae805010	0x335D9A010L	1	1	2	3	12509	512	1
0x2aacaf005010	0x334D9B010L	1	1	2	3	12493	768	1
0x2aacaf805010	0x335D9B010L	1	1	2	3	12509	768	1

Thread 1:

VirtualAddr	PhysicalAddr	Node	Channel	Rank	Bank	Row	Col	color
0x2aaab0000010	0x100198010L	0	1	2	3	11649	0	0
0x2aaab0800010	0x104785010L	0	0	2	0	11719	2304	0
0x2aaab1000010	0x157A48010L	0	1	2	5	4858	0	0
0x2aaab1800010	0x104786010L	0	0	2	0	11719	2560	0
0x2aaab2000010	0x111F41010L	0	0	2	4	3743	256	0
…….								
0x2aaab7800010	0x1210EF010L	0	1	3	5	12176	2816	0
0x2aaab8000010	0x12EF56010L	0	1	2	6	4207	2560	0
0x2aaab8800010	0x12A152010L	0	1	2	6	4129	512	0
0x2aaab9000010	0x124F31010L	0	1	3	2	4047	256	0
0x2aaab9800010	0x12A153010L	0	1	2	6	4129	768	0
……..								
0x2aab2c800010	0x32AC87010L	1	1	2	0	12332	2816	1
0x2aab2d000010	0x32698D010L	1	0	2	1	12265	2304	1
0x2aab2d800010	0x32ACC6010L	1	0	2	4	12332	2560	1
0x2aab2e000010	0x334985010L	1	0	2	0	12489	2304	1
0x2aab2e800010	0x3325A8010L	1	1	3	1	12453	0	1
0x2aab2f000010	0x32EF08010L	1	1	2	1	4207	0	1
0x2aab2f800010	0x3325A9010L	1	1	3	1	12453	256	1

Thread 2:

VirtualAddr	PhysicalAddr	Node	Channel	Rank	Bank	Row	Col	color
0x2aab30002010	0x157AA0010L	0	0	3	0	13050	0	0
0x2aab30802010	0x104658010L	0	0	2	7	3526	0	0
0x2aab31002010	0x21EF01010L	0	1	2	0	8047	256	0
0x2aab31802010	0x1046B6010L	0	1	3	2	11718	2560	0

0x2aab32002010	0x111CFD010L	0	0	3	7	11932	2304	0
…..								
0x2aab48002010	0x174482010L	0	1	2	0	13508	512	0
0x2aab48802010	0x171352010L	0	1	2	6	5267	512	0
0x2aab49002010	0x16BEE6010L	0	1	3	4	13374	2560	0
0x2aab49802010	0x171353010L	0	1	2	6	5267	768	0
0x2aab4a002010	0x17C165010L	0	1	3	4	5441	2304	0
…..								
0x2aabac802010	0x323EA5010L	1	0	3	0	12222	2304	1
0x2aabad002010	0x31F19D010L	1	1	2	3	12145	2304	1
0x2aabad802010	0x323EA6010L	1	0	3	0	12222	2560	1
0x2aabae002010	0x32FD0E010L	1	1	2	1	4221	2560	1
0x2aabae802010	0x32C420010L	1	1	3	0	4164	0	1
0x2aabaf002010	0x327756010L	1	1	2	6	4087	2560	1
0x2aabaf802010	0x32C421010L	1	1	3	0	4164	256	1

Thread 3:

VirtualAddr	PhysicalAddr	Node	Channel	Rank	Bank	Row	Col	color
0x2aabb0003010	0x21EFEC010L	0	0	3	5	16239	2048	0
0x2aabb0803010	0x104794010L	0	1	2	2	11719	2048	0
0x2aabb1003010	0x1000B2010L	0	1	3	2	11648	512	0
0x2aabb1803010	0x104795010L	0	1	2	2	11719	2304	0
0x2aabb2003010	0x111F6F010L	0	1	3	5	3743	2816	0
……								
0x2aabc2003010	0x15E8BE010L	0	1	3	3	13160	2560	0
0x2aabc2803010	0x15B2D7010L	0	1	2	6	13106	2816	0
0x2aabc3003010	0x1556A9010L	0	0	3	1	13014	256	0
0x2aabc3803010	0x15B2D8010L	0	1	2	7	13106	0	0
0x2aabc4003010	0x16800A010L	0	0	2	1	5120	512	0
……								
0x2aac2c803010	0x3273C7010L	1	1	2	4	12275	2816	1
0x2aac2d003010	0x3210F3010L	1	0	3	6	12176	768	1
0x2aac2d803010	0x3273C8010L	1	1	2	5	12275	0	1
0x2aac2e003010	0x330E77010L	1	1	3	6	4238	2816	1
0x2aac2e803010	0x32F44B010L	1	1	2	5	4212	768	1
0x2aac2f003010	0x328EEC010L	1	1	3	5	12302	2048	1
0x2aac2f803010	0x32F44C010L	1	1	2	5	4212	2048	1

From the data above, we can clearly find that there is memory bank contention between those threads. They would like to access same memory bank and get a much long latency. Besides, for one thread, it would access different memory node (memory controller). So the memory accessing latency is very high and unpredictable.

For the new coloring allocation based on memory controller, we get the following data:

Thread 1:

VirtualAddr	PhysicalAddr	Node	Channel	Rank	Bank	Row	Col	color
0x2b0ee6fcf010	0x15D3A2010L	0	1	3	0	13139	512	0
0x2b0ee7fcf010	0x15EFC2010L	0	1	2	4	13167	512	0
0x2b0ee8fcf010	0x21E093010L	0	0	2	2	16224	768	0
0x2b0ee9fcf010	0x1E4EAC010L	0	0	3	1	15310	2048	0
0x2b0eeafcf010	0x1713B3010L	0	0	3	2	13459	768	0
0x2b0eebfcf010	0x21E094010L	0	0	2	2	16224	2048	0
0x2b0eecfcf010	0x195D93010L	0	1	2	2	14045	768	0
…….								
0x2b0fe2fcf010	0x192C48010L	0	1	2	5	5804	0	0
0x2b0fe3fcf010	0x11D449010L	0	1	2	5	3924	256	0
0x2b0fe4fcf010	0x1B0C49010L	0	1	2	5	6284	256	0
0x2b0fe5fcf010	0x192C4A010L	0	1	2	5	5804	512	0

Thread 2:

VirtualAddr	PhysicalAddr	Node	Channel	Rank	Bank	Row	Col	color
0x2b0fe7fd1010	0x3AC911010L	1	0	2	2	6217	256	1
0x2b0fe8fd1010	0x2B9C85010L	1	1	2	0	10524	2304	1
0x2b0fe9fd1010	0x30E1C2010L	1	1	2	4	11873	512	1
0x2b0feafd1010	0x30EBED010L	1	0	3	5	11883	2304	1
0x2b0febfd1010	0x2B9C86010L	1	1	2	0	10524	2560	1
0x2b0fecfd1010	0x253E4D010L	1	1	2	5	702	2304	1
0x2b0fedfd1010	0x30EBEE010L	1	0	3	5	11883	2560	1
…….								
0x2b10e2fd1010	0x357B41010L	1	0	2	4	4859	256	1
0x2b10e3fd1010	0x359F42010L	1	0	2	4	4895	512	1
0x2b10e4fd1010	0x355F42010L	1	0	2	4	4831	512	1
0x2b10e5fd1010	0x357B43010L	1	0	2	4	4859	768	1

Thread 3:

VirtualAddr	PhysicalAddr	Node	Channel	Rank	Bank	Row	Col	color
0x2b10e7fd2010	0x62F5F2010L	3	1	3	6	8309	512	3
0x2b10e8fd2010	0x6A9FBD010L	3	0	3	3	10271	2304	3
0x2b10e9fd2010	0x81C5BE010L	3	0	3	3	16197	2560	3
0x2b10eafd2010	0x631FBD010L	3	0	3	3	8351	2304	3
0x2b10ebfd2010	0x6A9FBE010L	3	0	3	3	10271	2560	3
0x2b10ecfd2010	0x633FBD010L	3	0	3	3	8383	2304	3
0x2b10edfd2010	0x631FBE010L	3	0	3	3	8351	2560	3
…….								
0x2b11e2fd2010	0x75D3B5010L	3	0	3	2	13139	2304	3
0x2b11e3fd2010	0x75B3B6010L	3	0	3	2	13107	2560	3
0x2b11e4fd2010	0x75F3B6010L	3	0	3	2	13171	2560	3
0x2b11e5fd2010	0x75D3B7010L	3	0	3	2	13139	2816	3

Thread 4:

VirtualAddr	PhysicalAddr	Node	Channel	Rank	Bank	Row	Col	color
0x2b11e7fd3010	0x53C309010L	2	1	2	1	4419	256	2
0x2b11e8fd3010	0x5F4378010L	2	0	3	7	7363	0	2
0x2b11e9fd3010	0x5F5555010L	2	1	2	6	7381	2304	2
0x2b11eafd3010	0x5FBAC0010L	2	0	2	4	15674	0	2
0x2b11ebfd3010	0x5F4379010L	2	0	3	7	7363	256	2
0x2b11ecfd3010	0x5EB608010L	2	0	2	1	7222	0	2
0x2b11edfd3010	0x5FBAC1010L	2	0	2	4	15674	256	2
……								
0x2b12e2fd3010	0x481A7C010L	2	1	3	7	1434	2048	2
0x2b12e3fd3010	0x47EE7D010L	2	1	3	7	1390	2304	2
0x2b12e4fd3010	0x483A7D010L	2	1	3	7	1466	2304	2
0x2b12e5fd3010	0x481A7E010L	2	1	3	7	1434	2560	2

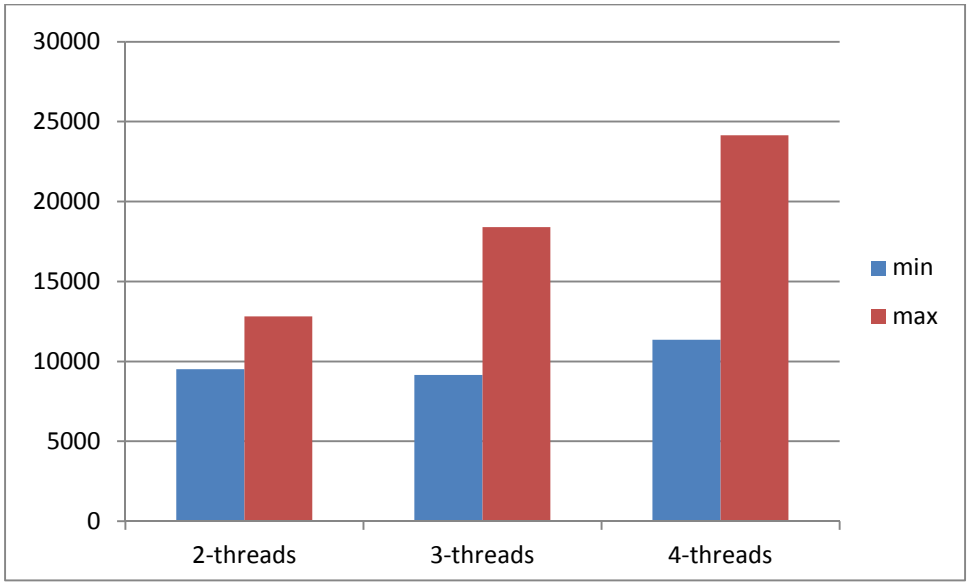
From the data above, we can clearly find that there is no memory bank contention between those threads. They are assigned different color memory and access different memory bank. Due to each thread only access the colored memory and the latency of accessing is lower and predictable.

The result also shows that the coloring allocation based on memory controller is correct and it could allocate the memory space to each thread based on memory controller successfully.

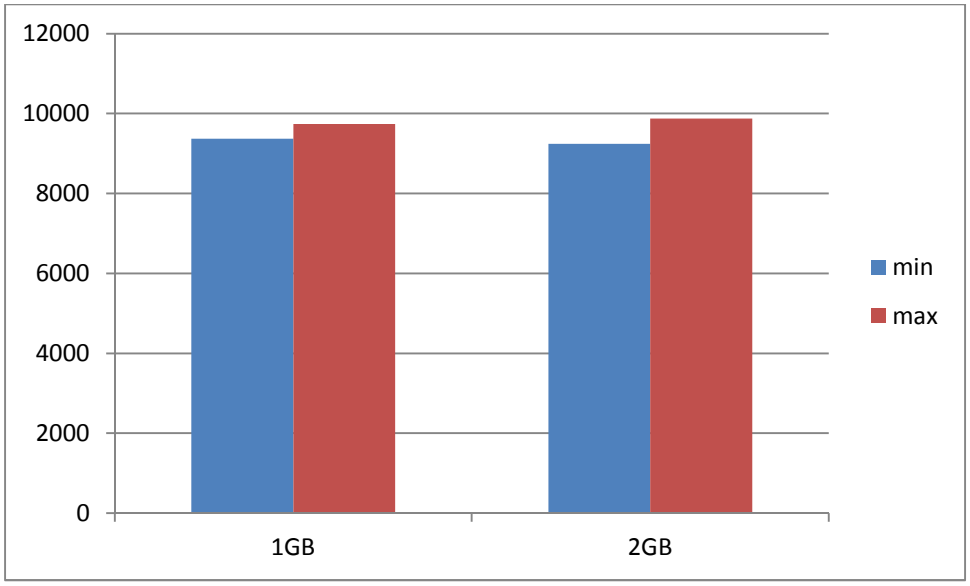
(2) Memory accessing latency between different threads number

The program creates different number of threads and tests the memory accessing latency under general and coloring allocation. The test program creates 2,3,4 threads respectively and allocates 2 GB memory to each thread.

The memory accessing latency under general allocation as following:



The memory accessing latency under coloring allocation based on memory controller as following:



From the graph above, we can clearly see that there is a big gap between the maximum latency and the minimum latency under the general allocation. Especially, there is more tasks in the program, there is larger gap. We can see the maximum latency is 34.7% more than minimum latency when there are 2 threads. However, it enlarges to almost 112.6% when there are 4 threads in the program. So in the real time system, it is very bad because the program can't predict the memory accessing latency exactly.

However, the problem is not existed when we use the coloring allocation based on memory controller. The latencies are almost same whatever how many threads. And the maximum is just 6%-8% more than minimum latency. So the system can predict the memory accessing latency exactly under the coloring allocation. It is very important for real time system.

(3) Memory accessing latency between different memory size allocated

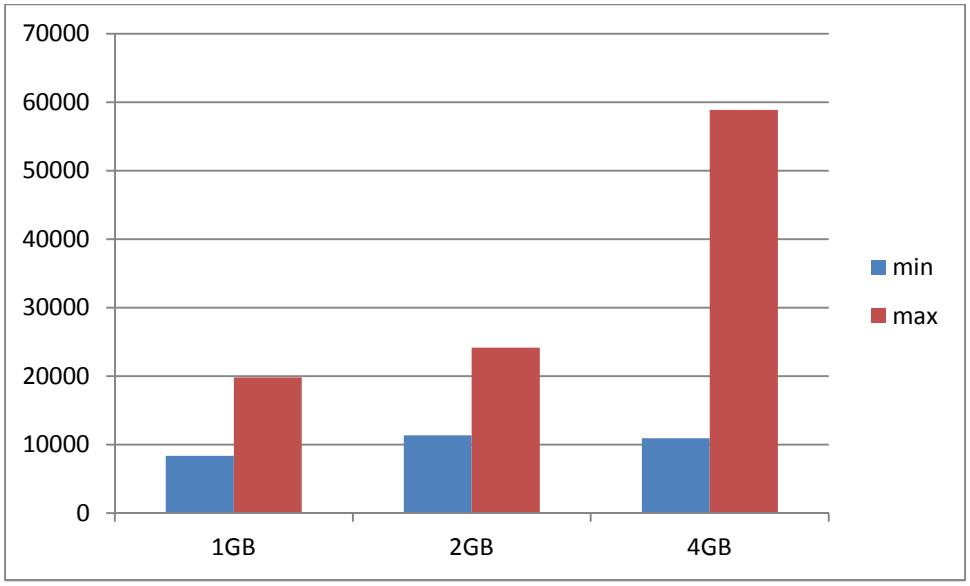
The program creates 4 threads and allocates them with different memory size. I test the memory accessing latency both under general and coloring allocation.

The test program creates 4 threads and allocates 1 GB memory for each thread.

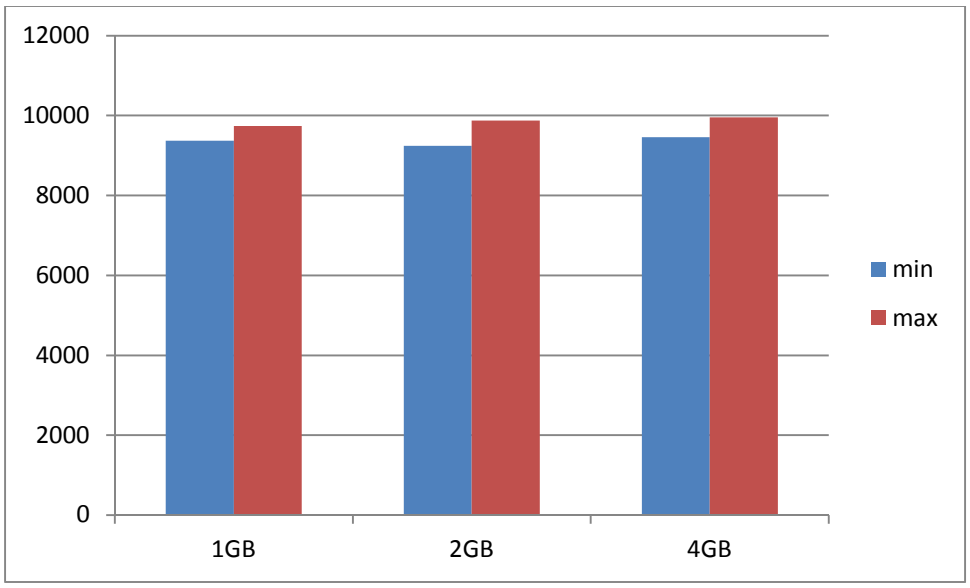
The test program creates 4 threads and allocates 2 GB memory for each thread.

The test program creates 4 threads and allocates 4 GB memory for each thread.

The memory accessing latency under general allocation as following:



The memory accessing latency under coloring allocation based on memory controller as following:

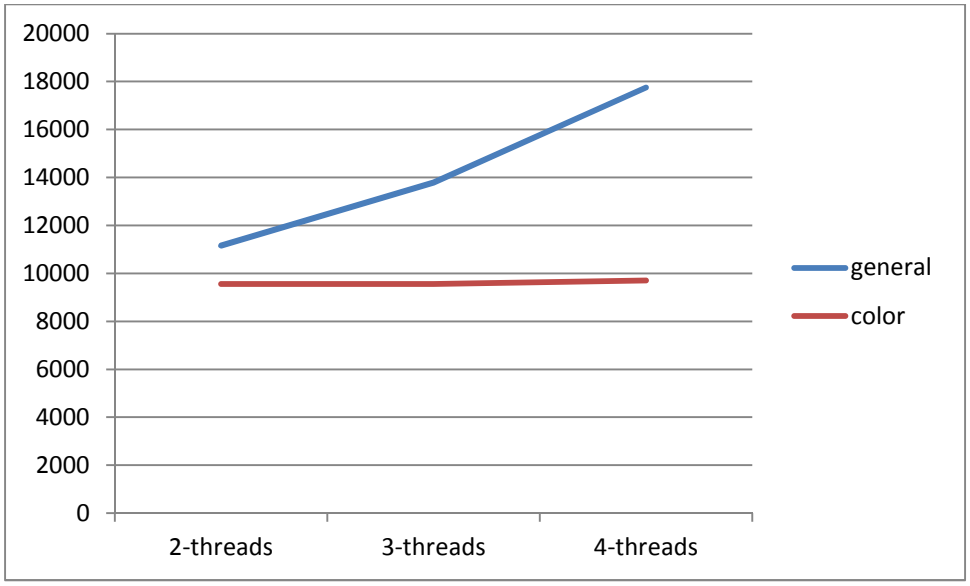


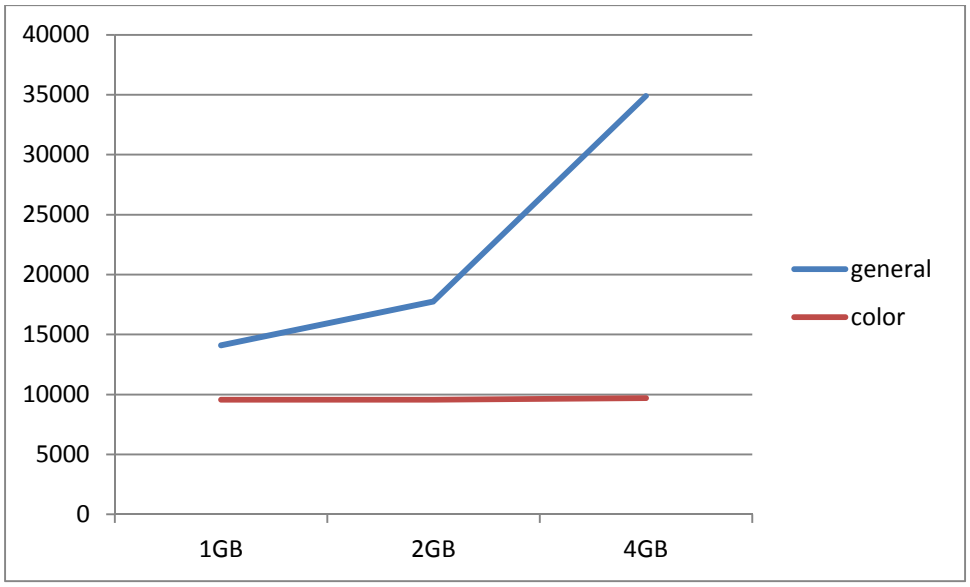
From the graph above, we can clearly see that there is a big gap between the maximum latency and the minimum latency under the general allocation. Especially, task allocates more memory space in the program, there is larger gap. We can see the maximum latency is 136.7% more than minimum latency when the task allocates 1 GB memory space. However, it enlarges to almost 437% when the task allocates 4 GB memory space. So in the real time system, it is very bad because the program can't predict the memory accessing latency exactly.

However, the problem is not existed when we use the coloring allocation based on memory controller. The latencies are almost same whatever how much memory space allocated. And the maximum is just 6%-8% more than minimum latency. So the system can predict the memory accessing latency exactly under the coloring allocation. It is very important for real time system.

(4) Better performance

Here, we can compare the average memory accessing latency under general and coloring allocation.





We can clearly find that using coloring allocation based on memory controller get a much better performance than general allocation.

Conclusion

Due there are several memory controllers in the modern CPU, the memory accessing latency is very hard to predict and the WECT is also not exactly in real time system. This project proposes a new coloring allocation algorithm to solve the memory accessing latency prediction problem in modern CPU. For the modern CPU, the new coloring allocation algorithm color main memory based on the memory controller. Each task in real time system will be assigned the different colored memory bank. As the result, we also show the correctness and latency predictable of this algorithm. So we can apply modern CPU (multiple memory controllers) in real time system by using the coloring allocation. Additionally, using the color allocation could also get better performance of memory accessing latency than general allocation.

Opening problem

1. Modify the algorithm and the kernel in the Linux. We still need to modify the allocation to make the tasks could access the remote memory controller.
2. Try to find whether there is any other reason for the latency
3. Try to realize dynamic coloring memory allocation.
4. Consider the memory fragment problem when we assign the colored memory bank to task.
5. Now, the coloring allocation only supports at most 4 tasks. We need to modify it to support more tasks.

Reference

[1] TILEPROCESSOR ARCHITECTURE OVERVIEW FOR THE TILEPROSERIES

[2] Christopher Zimmer and Frank Mueller. Low Contention Mapping of Real-Time Tasks onto a TilePro 64 Core Processor.

[3] Balasubramanya Bhat and Frank Mueller. Making DRAM Refresh Predictable

[4] Zheng Pei Wu, Yogen Krish, and Rodolfo Pellizzoni. Worst Case Analysis of DRAM Latency in Multi-Requestor Systems

[5] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo and Lui Sha. Memory Access Control in Multiprocessor for Real-time Systems with Mixed Criticality

[6] Wei Wang, Tanima Dey, Jack W. Davidson, and Mary Lou Soffa. DraMon: Predicting Memory Bandwidth Usage of Multi-threaded Programs with High Accuracy and Low Overhead

[7] Noriaki Suzuki, Hyoseung Kim, Dionisio de Niz. Coordinated Bank and Cache Coloring for Temporal Protection of Memory Accesses

[8] Hyoseung Kim, Dionisio de Niz, Bjorn Andersson. Bounding Memory Interference Delay in COTS-based Multi-Core Systems.

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

* The latency data in this document is the totally time when CPU access physical memory 524288 times.