CSC 714 Real Time Systems
Project Webpage: http://www4.ncsu.edu/~abahman/CSC714/

CSC 714 Project Progress Report 2

Analyzing the Effect of Predictability of Memory References on WCET

Amir Bahmani(abahman@ncsu.edu), Vishwanathan Chandru(vchandr6@ncsu.edu)

Objective:

To develop a colored malloc which colors MC and allocates memory according the colors based on various parameters like proximity to the tile, size of allocation, network contention etc.

Task Status:

1) **Ramp up on Tilera architecture and APIs**

Owner(s) : Both

Targeted completion : 24th March 2014

Status : Completed

Description : In this task, we were supposed to go through the Tilera architecture and APIs. We were successfully able to run the applications with the NoC library but for some reason couldn't run the applications without the shepherd running. Hence we will stick to using NoC library. We were successfully able to configure the accounts.

Open actions: PCI examples not running. Need to figure out why they are not working but its **low priority**.

2) **Figuring out base parameters for colored malloc**

Owner(s) : Vishwanathan

Targeted Completion: 24th March 2014

Status : Completed

Description : I started out looking into it to figure out what parameters could consider for color based allocation. I did some basic investigation to understand and figure out what can and cannot be done. I went through the architecture documentations of Tilera and hypervisor internals. The focus was on two key parameters proximity and striping. For proximity, I found that there are calls to figure out the closest MC. Hence decided we can consider proximity as one of the factors. The second factor is how we strip the address range. There were two options, either we do it based on virtual address or we do it on basis of physical address. I decided to go for virtual address basis as it will contiguous and will hide the effects of paging as the paging causes the physical address may change continuously and may result into clogging of a MC. The striping will be based on size of allocation in question. If it is small we could go for colouring in an interleaved fashion, if it is a huge allocation than we do it in a block interleaved fashion so as to ensure minimum contention and parallelization of memory access. We plan to try both the approaches with varying allocations and observer varying latencies.

Open Actions: None regarding implementation as the goal was just to analyze the feasibility of parameters

3) **Figure out way to remap the memory accesses**

Owner(s) : Vishwanathan

Targeted Completion: 31st March 2014

Status : **Deferred**

Description : The aim of this task was to figure out dynamic remapping of memory accesses. I started off with the investigating the possibility 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. Then as per the suggestion from Professor, this was investigated from a different angle. Instead of remapping on the fly, if we could do the loading of pages into memory adhering to the stripping and controller assignment done at virtual address level, the task will become much easier and effective. During the investigation 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. ***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, hence this task is deferred.***

Open Problems:

1) Page table functions are identified and corresponding interrupt handlers are located. Need to understand the flow to tap onto those functions and modify page loading.

2) Need to figure out how to build and deploy the modified Linux libraries. I could locate an android 3.0 build and a master's project involving porting Barrel Fish onto Tilera. Interestingly these involve downcall modifications. Given that and source code for hypervisor we could define a design and implementation for colored malloc.

3) Need to understand the exact control flow once the control enters the Linux kernel so as to modify allocations. If needed get in touch with the following people:

-> People in the Barrel Fish port

-> Try reaching out to Tilera or some forum to understand the downcall handling for page faults.

4) **Figure out a way to figure out the corresponding MC from physical address**

Owner(s) : Vishwanathan

Target Completion: 31st March 2014

Status : ***In Progress(Almost Done)***

Description : As a part of this task, it was required to identify the translation from VA to PA and from PA decode which MC to assign to. The translation mechanism using page map and page tables were understood and given the fact that downcall mechanism is used for PF handling, there is a strong chance that same might work for Tilera board.

Updates:

It was found that the standard approach used in linux worked just fine. The VA uses 32 bit and physical address uses a globally shared 36 bit address. That is, it is possible to address a total of 64 GB memory. This address is provided by mdn packet which is converted to external 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:

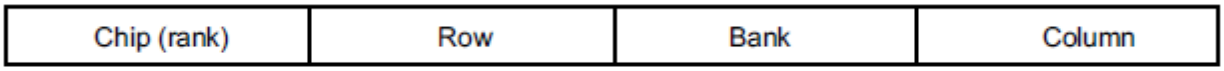

Figure 2-2: External Memory Address Format

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

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. Based on the /proc/pid/pagemap file the conversion to physical address was performed. There was one more challenge found during further investigation, the striping of memory pages. 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. Now the catch is the MDN provided address to external address conversion doesn't depend on striping. Given the page size of 64k, we convert a physical frame to physical address by multiplying it with 64k, but 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 behavior

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. I tried looking into the hypervisor code to figure out how the MDN 36 bit addresses are generated and at what point in the flow they are converted to external address but could not find anything useful. I tried to do inter-leaved page allocation on memory controllers as there is api support for that kind of allocation. This would give a better idea of the conversion to external address but even after modifying the hypervisor configuration file to suppress striping, there is some file which keeps enabling it and I am unable to locate that file. The reading of MC registers was a bit tricky to figure out the hashing values and memory configuration values, so eliminate that I switched to using tmc_alloc_map as it is possible to disable the L1 and L2 caching of data allocated using this method. Since L1 and L2 caching of data is disabled, it is not possible for DSM or hashing to come into action. It also has one additional advantage 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 colored malloc in the user space.

Open Actions :

1. Figure out how to switch off the memory striping so as to try out inter leaved page allocation. This might give us further hint into address translation.
2. The implementation of malloc is done assuming striping but to extend that implementation in a generic way more understanding is need on where and when exactly the MDN addresses are created and translated into physical address. As mentioned previously, I am observing some discrepancies in the results I am getting and documented behavior. This is needed to solve the observed discrepancies.
3. The longest possible external address is 29 bits and for addressing 16 GB of memory minimum of 34 bits is required. There is some missing link over here which needs to be figured out.
4. For some strange reasons which I am not able to figure out yet, I am not able to allocate memory more that 1.5 GB which given the 32 bit VA range is strange and not expected.
5. Need to sort out a mismatch between specs mentioned on the site and that extracted from the system.

5) **Memory mapping strategy**

Owner(s) : Vishwanathan

Target Completion: 7th April 2014

Status : Implementation Complete (Debugging pending)

Description : Since we are doing any kernel level modification to support and control dynamic allocation of pages, our approach is pretty restrictive. We create a memory pool of a huge chunk of memory and for each allocation performed over this chunk is done using the colouring scheme. We do the allocation using tmc_alloc_map disabling homing and caching. We then iterate through memory (at granularity depending upon fact whether its striped memory or not) and update the mapping capturing color of the particular segment. When it comes to allocation there are two possible approaches:

1) Whenever memory is requested from pool generate a random color, and try to allocate a memory of generated color. If not possible, allocate any free chunk. This approach is bit non uniform and could lead to exhaustion of a particular color.
2) This approach requires caching of the last allocated color. Whenever an allocation is requested, we increment the cached color by 1 and search for the memory of that particular color. If found well and good, but if not found allot whatever free slot was found and update the cached color with the color of allocated memory.

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.

Open Actions:

1) Unable to figure out a way to turn off memory striping.
2) It depends on color being determined accurately of which I am not sure yet.

6) **Strategy for contention analysis**

Owner(s) : Vishwanathan

Target Completion: 7th April 2014

Status : Implemented(Debugging in Progress)

Description : As a part of this, I plan to measure the impact of NoC contention using repeated memory accesses considering only RAW, WAR and WAW dependency. This has to be done to ensure that there is no caching in the L1 and L2 cache. Since I decided to go with tmc_alloc_map, it has parameters to disable caching and homing of data. The plan to test is to use pthreads to emulate tasks. We allocate a huge chunk of memory and determine the number of elements of size 8k/page size it can hold. After determination of the number of elements we allocate the chunks and elements are distributed among threads. Now the threads repeatedly access the memory elements in a loop depending upon number of times we want each element to be traversed. We keep track of the cycles spent in traversal of allotted elements given number of times and perform the averaging at the end. So this way we end up measuring latency using cycles. To figure out latency with standard malloc we can just replace the allocation of elements from the pool to standard malloc allocation.

Open Actions:

1) Unable to figure out a way to turn off memory striping.
2) It depends on color being determined accurately of which I am not sure yet.

7) **Malloc implementation**

Owner(s) : Both

Target Completion: 14th April 2014

Status : Implemented (Debugging in Progress)

Description : Implementation of pooled memory and colored malloc is done. Debugging is in progress.

8) **Test Cases Design and Implementation**

Owner(s) : Both

Target Completion: 14th April 2014

Status : Implemented (Debugging in Progress)

Description : Test case is written but there seems to be some issues with colored malloc itself which needs to be fixed before test case can be frozen.

9) **Task Mapping Algorithm and implementation**

Owner(s) : Amir

Target Completion: 21st April 2014

Status : To Do

Description : This task is part of our secondary objective under which we investigate optimal placement of inter dependent task. It will be attempted once we are done with primary objective.

10) **Task Mapping algorithm validation**

Owner(s) : Vishwanathan
Target Completion: 25th April 2014
Status : To Do
Description : Test development and validation of task mapping

11) **Presentation**

Owner(s) : Amir
Target Completion: 27th April 2014
Status : To Do
Description : Demo of the malloc prototype and task mapping(if task mapping done).

12) **Final Report**

Owner(s) : Both
Target Completion: 1st May 2014
Status : To Do
Description : Final report submission

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