Multicore Cache and TLB coloring on Tegra3

Final Project Report

(CSC 714 – Spring 2014)

Professor: Dr. Frank Mueller

URL for the project: https://sites.google.com/a/ncsu.edu/multicore-cache-and-TLB-coloring/

Project Members

(In alphabetical order)

Payal Godhani           pgodhan@ncsu.edu

Shrinivas Anand Panchamukhi  spancha@ncsu.edu

Yasaswini Jyothi Gownivaripalli   ygowniv@ncsu.edu
Links

Yasaswini's report ------ page 3
Shrinivas's report ------ page 7
Payal's report --------- page 17
Abstract

In multi-core systems, predictability of execution has been a concerning problem due to interaction among the cores. In absence of precisely characterizing such interactions, worst case execution time is measured with pessimistic assumptions. These assumptions will negate the extra processing power gained through multi-cores. Our attempt in this project is three fold.

- First is to port cache coloring scheme developed by UNC onto the ARM based Tegra3 Kayla board and measure the improvements
- Second is to port TLB coloring developed at NC State onto ARM based Tegra3 Kayla board and measure the improvements
- Third is to integrate TLB coloring on top of cache coloring and measure cumulative improvement
Following pages describe the report of Yasawini

<table>
<thead>
<tr>
<th>Project Member: Yasaswini</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task Description: Porting colored litmus for cache coloring on Tegra3</td>
</tr>
</tbody>
</table>

### L2 cache configuration on Kayla Tegra3:
- Cache size: 1MB, Line_size:32
- 8 ways, 4096 sets
- Litmus color mask: Bits 17 – 13

### 1. PAPI setup on Tegra3
- PAPI along with it’s dependent libraries (libpfm) are installed on Tegra3
- Ran test cases of PAPI on Tegra3
- Cross development environment and NFS for Tegra3 is setup
- Demonstrated to group members kernel compilation, bootable drive creation, library installation and NFS setup on Tegra3

### 2. Patch-up Linux kernel with colored-malloc implementation of Litmus Rt (Completed)
- Identified the kernel differences between UNC litmus kernel, Tegra3 kernel and Vanilla kernels
- Modified Tegra3 kernel along the lines of UNC litmus cache coloring implementation
- Resolved compilation issues and generated kernel image

### 3. Study changes required for Tegra3 cache coloring, identify and resolve implementation issues during booting of kernel (Completed)
- UNC-litmus-linux-3.0.0 ported to Tegra3-linux-3.1.10 and booted on Kayla board
- Validated the modifications of Tegra3-litmus-linux-3.1.10 against UNC-litmus-linux-3.1.10
- Observed – Modifications in litmus data-structures of UNC-litmus-linux-3.1.10 compared to UNC-litmus-linux-3.0.0
- Compared implementations of UNC litmus and Tegra3 kernel, isolated the problems and resolved
- Booted litmus patched Linux-3.1.10 on Tegra3

### 4. Run Litmus Rt patched kernel, liblitmus based applications (Completed)
**(colored-liblitmus porting was extended by 5 working days due to un-availability of source)**
- Cache coloring feature added as an add-on compatible feature of menuconfig in kernel
- Resolved the issue of kernel hanging while scheduling litmus tasks
- Ported liblitmus for Tegra3-litmus-Linux-3.1.10 by modifying data structures in header files
- Installed libgsl on Tegra3-litmus-Linux (liblitmus has dependency on it)
- Installed feather-trace-tools on Tegra3-litmus-kernel
- Compiled liblitmus based applications and tested cache coloring API
- Identified and resolved the invalid memory access by litmus code while accessing proc filesystem
- Tested cache coloring API successfully

5. Run experiments and collect the results (Completed)

root@seco-gpu-devkit:~/yasaswini/liblitmus_coloring# cat /proc/sys/litmus/color/cache_info

Cache size : 1048576 B
Line size : 32 B
Page size : 4096 B
Ways : 8
Sets : 4096
Colors : 32

root@seco-gpu-devkit:~/yasaswini/liblitmus_coloring# ./setsched GSN-EDF

root@seco-gpu-devkit:~/yasaswini/liblitmus_coloring# ./colortest

tests passed.

root@seco-gpu-devkit:~/yasaswini/liblitmus_coloring# ./armsinglepage 2 4
6. Colored litmus project source and documentation

- Project source and documentation checked-in to CVS under ygowniv/tegra3_litmus
- Documentation is present in Readme or Install files along with respective source directories of kernel, library and other packages
- URL for the source in CVS: http://optout.csc.ncsu.edu:7467/viewvc/ygowniv/tegra3_litmus/

7. Open problems

- Coloring other levels of cache (ex: L1 cache on Tegra3)
- Implementing memory coloring along with the cache coloring
Following pages describe the report of Shrinivas

Problem statement / motivation for TLB coloring

Data translation look aside buffer (DTLB) presents a source of unpredictability for real time systems. Figure 1 shows a 2 way set associative DTLB with 4 sets. Two tasks T1 and T2 running on the same core request memory from the standard heap allocator malloc. malloc could return virtual addresses such that they map to the same DTLB set, causing the tasks to replace each other's page mappings in the TLB.

![Diagram of 2 way set associative TLB](image)

### Proposed solution

We use the concept of page coloring to bring in predictability for real time tasks which use dynamic memory allocation. We color virtual pages in such a manner that two virtual pages with different color do not map to the same DTLB set. The concept is explained using an example. Consider a virtual address space with 64 pages and a 4-way set associative DTLB supporting 64 entries as shown in Figure 2. The addresses on the left-hand side of the figure represent the base virtual address for the corresponding virtual page. For this particular example, assume that the page size is 4kB. Since the page size is 4kB, bits 0-11 of the virtual address will determine the offset within the page. In the above example,
the DTLB has 16 sets. Hence we need 4 bits to identify the DTLB set. Bits 12-15 of the virtual address will determine which DTLB set the virtual page will map to. As shown in Figure 2, page 0 will map to set 0. The translation for page 0 could be stored in any one of the 4 ways. For simplicity, let us assume that the DTLB is empty initially. We fill the entries from left to right in each set. Continuing our mapping further,

page 1 will map to set 1, page 2 will map to set 2 and so on until page 15 which maps to set 15. Pages 16 to 63 will rap around, i.e. page 16 will map to set 0, page 17 will map to set 1 and the other pages will map in a similar fashion.

![Data Translation lookaside buffer](image)

We color pages 0, 16, 32, 48 with the same color (red in this example) because all of them map to the same DTLB set. Similarly pages 1, 17, 33, 49 are colored blue and so on. We can see that no two virtual pages with different color can map to the same DTLB set. Each DTLB entry holds a translation for a 4kB virtual page. Since each DTLB set is given one color, the maximum contiguous virtual address space one can allocate of a particular color is 4kB.
Design of tlb_malloc for Tegra 3

The NVIDIA Tegra 3 board has a fully associative L1 DTLB supporting 32 entries and a 2-way set associative L2 DTLB supporting 128 entries. The coloring scheme can only be applied to set associative DTLBs and hence we perform coloring for L2 DTLB. There is a peculiarity about the L2 DTLB on Tegra 3. It supports a fully associative array supporting four entries at the L2 DTLB which are lockable. When no entries are locked in this array, these four entries are also used in addition to the 128 entries thereby increasing the size of the L2 DTLB. Effectively the L2 DTLB supports 132 entries and the lockable array provides space for the page mappings which spill over. These four lockable entries can also be viewed as a victim cache. These four entries pose a design problem to our coloring method. To alleviate the problem, we consider these four lockable entries as additional instances of a particular color.

Implementation of tlb_malloc for Tegra 3

API details

We refer to the routine which initializes our heap allocator as tlb_malloc_init, the heap allocator as tlb_malloc and the deallocator as tlb_free. Table I describes the parameters for each of these routines. These three routines are exposed as library functions to user space applications. As part of initialization tlb malloc init sets aside for each task a virtual address space of 4kB * 4k_dtlb_sets + huge_page_size * huge_page_dtlb_sets. Additional memory may be needed to handle page boundary alignment. Our heap allocator performs allocations from this virtual address space that is set aside.

<table>
<thead>
<tr>
<th>Function name</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tlb_malloc_init</td>
<td>4k_dtlb_sets</td>
<td>Total number of sets in the DTLB which handles 4k translations</td>
</tr>
<tr>
<td></td>
<td>4k_dtlb_assoc</td>
<td>Associativity of the DTLB which handles 4k translations</td>
</tr>
<tr>
<td></td>
<td>Huge_page_dtlb_sets</td>
<td>Total number of sets in the DTLB which handles huge page translations</td>
</tr>
<tr>
<td></td>
<td>Huge_page_dtlb_assoc</td>
<td>Associativity of the DTLB which handles huge page translations</td>
</tr>
<tr>
<td></td>
<td>4k_dtlb_sets_for_array</td>
<td>Number of DTLB sets in the 4k DTLB to be kept reserved for contiguous allocations</td>
</tr>
</tbody>
</table>
Table I

Depending on the number of bytes requested, tlb_malloc will call one of the functions listed in Table II. LEN_BYTES refers to the number of bytes the allocator uses to store the size of the allocation. huge_page_size is the size of a huge_page. Similarly tlb_free will call one the functions listed Table III depending on the size of the allocation pointed by ptr.

<table>
<thead>
<tr>
<th>Function name</th>
<th>Allocation size</th>
</tr>
</thead>
<tbody>
<tr>
<td>tlb_malloc_small</td>
<td>( &lt;= 4kB - LEN_BYTES )</td>
</tr>
<tr>
<td>tlb_malloc_large</td>
<td>( &gt;4kB - LEN_BYTES ) and ( &lt;= \left( (4k_dtlb_sets_for_array \times 4kB) - LEN_BYTES \right) )</td>
</tr>
<tr>
<td>tlb_malloc_huge_small</td>
<td>( &gt;((4k_dtlb_sets_for_array \times 4kB) - LEN_BYTES) ) and ( &lt;= \left( huge_page_size - LEN_BYTES \right) )</td>
</tr>
<tr>
<td>tlb_malloc_huge_large</td>
<td>( &gt;\left( huge_page_size - LEN_BYTES \right) ) and ( &lt;= \left( \left( huge_page_size \times dtlb_sets_for_array \right) - huge_page_size \right) - LEN_BYTES )</td>
</tr>
</tbody>
</table>

Table II
Table III

<table>
<thead>
<tr>
<th>Function name</th>
<th>Allocation size</th>
</tr>
</thead>
<tbody>
<tr>
<td>tlb_free_small</td>
<td>$\leq 4kB$</td>
</tr>
<tr>
<td>tlb_free_large</td>
<td>$&gt;4kB$ and $\leq (4k_{dlb_sets_for_array} \times 4kB)$</td>
</tr>
<tr>
<td>tlb_free_huge_small</td>
<td>$&gt;(4k_{dlb_sets_for_array} \times 4kB)$ and $\leq (\text{huge_page_size})$</td>
</tr>
<tr>
<td>tlb_free_huge_large</td>
<td>$&gt;(\text{huge_page_size})$ and $\leq (\text{huge_page_dtlb_sets_for_array} \times \text{huge_page_size})$</td>
</tr>
</tbody>
</table>

Heap allocator and deallocator algorithms

Algorithm 1 shows the pseudo code for our heap allocator. Each of the functions listed in Table II invokes this algorithm. The parameter type is used to identify which function listed in Table II invokes this algorithm. Line 3 sets free list to the appropriate list depending on type and color. The function on line 5 is responsible for walking through the free list to find a suitable memory block and returning the starting address of the block which can be used by the user. It is also responsible for storing the allocation size in LEN BYTES preceding the returned starting address.

Algorithm 2 shows the pseudo code for our heap deallocator. Since we store the length of the allocation in LEN BYTES preceding ptr, lines 2 & 3 get the length of the allocation and the base address of the block referenced by ptr. The function on line 5 adds the memory block back to the appropriate free list.
Experiments and results

The following sub sections describe two experiments conducted. The first experiment describes the best case while the second one describes the worst case. For both the cases the following task set is assumed. $T(\text{phase, period, execution}) = \{ T1(1\text{ms},2\text{ms},0.4\text{ms}), T2(0\text{ms},16\text{ms},8\text{ms})\}$. In each of the experiment we use tlb_malloc and each thread allocates 64 pages and job0 refers to the warm up job.

**Best case**

The best case situation is when 64 pages of each thread map to non conflicting sets in the L2 DTLB.

<table>
<thead>
<tr>
<th>Hyper period number</th>
<th>Job number</th>
<th>Main_DTLB_miss_cycles</th>
<th>Micro_DTLB_miss_cycles</th>
<th>time in micro seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>For thread 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>-------</td>
<td>-------------------</td>
<td>-------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>0</td>
<td>1740</td>
<td>352.3333333</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>827.3333333</td>
<td>27076.3333333</td>
<td>326.6666667</td>
<td></td>
</tr>
<tr>
<td></td>
<td>647</td>
<td>27307.3333333</td>
<td>323.6666667</td>
<td></td>
</tr>
<tr>
<td></td>
<td>285.6666667</td>
<td>27157.3333333</td>
<td>322</td>
<td></td>
</tr>
<tr>
<td></td>
<td>658</td>
<td>27127.3333333</td>
<td>322</td>
<td></td>
</tr>
<tr>
<td></td>
<td>567.3333333</td>
<td>27108</td>
<td>322</td>
<td></td>
</tr>
<tr>
<td></td>
<td>399.6666667</td>
<td>27018.66667</td>
<td>321</td>
<td></td>
</tr>
<tr>
<td></td>
<td>597.6666667</td>
<td>27154</td>
<td>324</td>
<td></td>
</tr>
<tr>
<td></td>
<td>470.3333333</td>
<td>27354</td>
<td>321.3333333</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>512.6666667</td>
<td>27188.66667</td>
<td>321.6666667</td>
<td></td>
</tr>
<tr>
<td></td>
<td>535</td>
<td>27216.3333333</td>
<td>321.3333333</td>
<td></td>
</tr>
<tr>
<td></td>
<td>470.3333333</td>
<td>27222.66667</td>
<td>321</td>
<td></td>
</tr>
<tr>
<td></td>
<td>470</td>
<td>27130.3333333</td>
<td>321.3333333</td>
<td></td>
</tr>
<tr>
<td></td>
<td>592.6666667</td>
<td>27140.33333</td>
<td>322.6666667</td>
<td></td>
</tr>
<tr>
<td></td>
<td>628.6666667</td>
<td>27139.33333</td>
<td>323.6666667</td>
<td></td>
</tr>
<tr>
<td></td>
<td>304</td>
<td>27195.66667</td>
<td>321.6666667</td>
<td></td>
</tr>
<tr>
<td></td>
<td>373</td>
<td>27125.33333</td>
<td>321.3333333</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>330</td>
<td>27110</td>
<td>321</td>
<td></td>
</tr>
<tr>
<td></td>
<td>320</td>
<td>27108.66667</td>
<td>320.6666667</td>
<td></td>
</tr>
<tr>
<td></td>
<td>248</td>
<td>27114.66667</td>
<td>321</td>
<td></td>
</tr>
<tr>
<td></td>
<td>377.3333333</td>
<td>27140</td>
<td>321</td>
<td></td>
</tr>
<tr>
<td></td>
<td>252.3333333</td>
<td>27144</td>
<td>321</td>
<td></td>
</tr>
<tr>
<td></td>
<td>369</td>
<td>27154.33333</td>
<td>320.6666667</td>
<td></td>
</tr>
<tr>
<td></td>
<td>307.3333333</td>
<td>27178.33333</td>
<td>320.6666667</td>
<td></td>
</tr>
<tr>
<td></td>
<td>317.3333333</td>
<td>27179.33333</td>
<td>320.6666667</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>356.3333333</td>
<td>27041.66667</td>
<td>320.3333333</td>
<td></td>
</tr>
<tr>
<td></td>
<td>350.3333333</td>
<td>27067.33333</td>
<td>320.6666667</td>
<td></td>
</tr>
<tr>
<td></td>
<td>342.6666667</td>
<td>27212.33333</td>
<td>320.6666667</td>
<td></td>
</tr>
<tr>
<td></td>
<td>298.6666667</td>
<td>27143.66667</td>
<td>320.6666667</td>
<td></td>
</tr>
<tr>
<td></td>
<td>356.3333333</td>
<td>27008.66667</td>
<td>320.6666667</td>
<td></td>
</tr>
<tr>
<td></td>
<td>339</td>
<td>27282.33333</td>
<td>320.6666667</td>
<td></td>
</tr>
<tr>
<td></td>
<td>310.3333333</td>
<td>27067.66667</td>
<td>321</td>
<td></td>
</tr>
<tr>
<td></td>
<td>317.3333333</td>
<td>27054.33333</td>
<td>320.6666667</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>330</td>
<td>27022</td>
<td>320.6666667</td>
<td></td>
</tr>
<tr>
<td></td>
<td>383.3333333</td>
<td>27189</td>
<td>320.6666667</td>
<td></td>
</tr>
<tr>
<td></td>
<td>229.6666667</td>
<td>27148.66667</td>
<td>320.6666667</td>
<td></td>
</tr>
<tr>
<td></td>
<td>294.6666667</td>
<td>26990.66667</td>
<td>320.6666667</td>
<td></td>
</tr>
<tr>
<td></td>
<td>283.3333333</td>
<td>27045</td>
<td>321</td>
<td></td>
</tr>
<tr>
<td></td>
<td>299.6666667</td>
<td>27067.66667</td>
<td>320.6666667</td>
<td></td>
</tr>
<tr>
<td></td>
<td>313.6666667</td>
<td>27008.66667</td>
<td>320.6666667</td>
<td></td>
</tr>
<tr>
<td></td>
<td>334</td>
<td>27115</td>
<td>320.6666667</td>
<td></td>
</tr>
</tbody>
</table>
Table IV

For thread 2

<table>
<thead>
<tr>
<th>Hyper period number</th>
<th>Job number</th>
<th>Main_DTLB_miss_cycles</th>
<th>Micro_DTLB_miss_cycles</th>
<th>time in microseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1671.666667</td>
<td>326</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>707</td>
<td>64508.6667</td>
<td>7834.333</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>756</td>
<td>646461</td>
<td>7569.333</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>656.666667</td>
<td>645636</td>
<td>7523.667</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>515.666667</td>
<td>646628</td>
<td>7442.667</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>751</td>
<td>646132.6667</td>
<td>7646.667</td>
</tr>
</tbody>
</table>

Table V

Worst case

The worst case situation is when 64 pages of each thread map to conflicting sets in the L2 DTLB. We use tlb_malloc to ensure this by having each thread allocate the same set of colors.

For Thread 1

<table>
<thead>
<tr>
<th>Hyper period number</th>
<th>Job number</th>
<th>Main_DTLB_miss_cycles</th>
<th>Micro_DTLB_miss_cycles</th>
<th>time in microseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1985.666667</td>
<td>350.3333333</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>640.3333333</td>
<td>27218</td>
<td>332.3333333</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>634.666667</td>
<td>27140.333333</td>
<td>331</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>535.3333333</td>
<td>27124</td>
<td>329.666667</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>538.666667</td>
<td>27114.333333</td>
<td>9320.66667</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>2358.3333333</td>
<td>27755.66667</td>
<td>393</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>773</td>
<td>27278.66667</td>
<td>335.3333333</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>829.666667</td>
<td>27096</td>
<td>332.666667</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>349.3333333</td>
<td>27097.66667</td>
<td>329</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>470</td>
<td>27154</td>
<td>329</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>771</td>
<td>27210.66667</td>
<td>335.666667</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>528.666667</td>
<td>27103.66667</td>
<td>329.666667</td>
</tr>
<tr>
<td>Hyper period number</td>
<td>Job number</td>
<td>Main_DTLB_miss_cycles</td>
<td>Micro_DTLB_miss_cycles</td>
<td>time in micro seconds</td>
</tr>
<tr>
<td>---------------------</td>
<td>------------</td>
<td>-----------------------</td>
<td>------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>0</td>
<td>1463</td>
<td>325.6666667</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1486.666667</td>
<td>642059.3333</td>
<td>328.333333</td>
<td>8092.667</td>
</tr>
<tr>
<td>2</td>
<td>2288</td>
<td>643760</td>
<td>328.333333</td>
<td>8289</td>
</tr>
<tr>
<td>3</td>
<td>636.3333333</td>
<td>640272.3333</td>
<td>328.333333</td>
<td>7458.667</td>
</tr>
</tbody>
</table>
Tables IV - VII show the results for best case and worst case isolation scenarios for two tasks. Comparing Main_DTLB_miss_cycles of job1 in table IV with that of job1 in table VI we can see that the worst case has fewer stalls than best case. But if we compare job5 in table IV with job5 in table VI we can see that the best case performs better. Similar results are observed for other hyper periods. Tables V and VII show the experimental results for thread 2. We would have expected the response time of T2 to increase by five times the execution time of T1. This is because T1 preempts T2 five times in a hyper period. But this is not observed in table V and VII. Having the DTLB miss counts would have been very helpful in analyzing and improving the experiments. But the Tegra 3 board does not have performance monitoring registers capable of recording DTLB miss counts per level. Hence, with the experiments conducted it is difficult to reach a conclusion about task isolation.

Future work

One of the problems with the tlb_malloc implementation is that each task needs to call tlb_malloc_init. This needs to be changed to have tlb_malloc_init to be called only once by the main program which creates the tasks. The global variables in the tlb_malloc_init are per thread variables which needs to be changed to non thread specific variables. We also need to design additional experiments to show task isolation.
Integration of TLB coloring and Cache Coloring

Cache Coloring:

- **Problem:** In multi-core architecture the last level of cache is shared amongst all the processors as shown in figure 1. The inability to precisely predict whether a page will get a hit or miss in the cache introduces unpredictability due to which we cannot have tighter bounds on the worst case execution time as every time a page is considered as a cache miss.

![Diagram of cache hierarchy](image)

**Fig.1 : Last level of cache shared by all the cores on Tegra3**

As shown in figure 2. Physical pages belonging to different cores are mapping to the same cache set at run time. Since at run time we do not know whether physical page will get a hit or miss in cache it introduces unpredictability in the system. Using current technology, very pessimistic assumptions must be made regarding the utilization of these shared resources. The processing capacity lost to such pessimism can easily negate the impact of any additional cores.
Fig. 2. Pages belonging to different cores are mapping to the same cache set at run time.

**Solution:** The problem is addressed by considering several cache management schemes that utilize page coloring in some way. Under page coloring, pages of physical memory are assigned colors in a way that ensures that different colored cannot cause cache conflicts. Proper shared cache management can lessen WCETs and positively impact schedulability despite increased system overheads.

**Fig 3. Cache coloring**

Cache coloring specifications for Tegra 3:

Pages of physical memory are colored to ensure that differently colored pages cannot cause cache conflicts.

- **No of Colors used = 32**
- **Number of sets = 4096**
Number of ways = 8
Number of colors = 32
Sets per color = 128
Cache Line Size = 32 Bytes
L2 Shared Cache = 1 MB

Physical memory is subdivided into pages and each page is given a particular color. So color the first page.

Each page consists of 4 KB /32 Bits cache lines = 4096/32 = 128 lines

So first 128 cache lines will be assigned color 0.

Color 0    Sets 1 – 128
Color 1    Sets 129 – 256
Color 2    Sets 257 – 384 and so on.

Then after the 32\textsuperscript{nd} page, all 2\textsuperscript{12} sets will have been used and color assignments will wrap i.e. 33\textsuperscript{rd} page will map to the same cache sets as the first and we will reuse color 0. Moreover, two pages that are assigned two different colors will map to different cache sets and thus cannot conflict with each other in cache.

**TLB Coloring:**

TLB Coloring is explained above in detail.

**TLB Coloring with Cache Coloring:**

- **Problem:** In spite of adding coloring techniques at cache level and TLB level there exists no unified solution which avoids interference at both cache and TLB level. For example, there may...
be two pages mapping to two different cache sets in last level of cache but may be mapping to the same TLB set introducing interference in the system and hence unpredictability.

Fig 4. Two pages belonging to two different cache sets map to the same TLB set

Fig 5. Two pages belonging to two different TLB sets map to the same cache set

- **Solution:** To avoid a unified solution, interference should be avoided at both levels. Hence, two physical pages belonging to two different cache colors should map to two different tlb colors in order to avoid interference.
Fig 6. Two pages belonging to different cache colors should map to different TLB colors

**First Approach:**

**Step 1:** Allocate pages one at a time by using `color_malloc` of some specified color.

After requesting a physical page of some color, now we know that page x is having color y in physical memory.

**Step 2:** Examine the virtual address returned by `color_malloc`

Once we have the virtual address returned by `color_malloc`, from the virtual address bits we can determine which TLB set it is mapping to and hence which color because we assume that each TLB set is of different color.

**Step 3:** Store the mapping Physical page color mapping to Virtual page color mapping

Now we will build the free list of TLB based on this mapping. Suppose red color physical page is mapping to pink color TLB page we will create a free list of pink color in TLB where in virtual address corresponding to red color physical page will be one of its entry.

**Following are the three scenarios which we have to consider:**

**Scenario 1:** Now if we encounter a green color physical page mapping to same pink color in TLB, we will not ask for the green color since it will lead to conflict in TLB.

**Scenario 2:** Supposing, if we ask for a red color page from `color_malloc` and it maps to violet color in TLB we will not proceed with this allocation.

**Scenario 3:** If we get a red color page with a virtual address which maps to pink color TLB set, we will proceed and add this page to free list of pink in TLB ensuring that no conflicts will color at any level.
Problem: Cannot allocate more than 32 pages by using color_malloc.

Approach 1:

Implementation:

Step 1: Created color control structure by specifying number of pages to allocate and number of colors to be used.

```
struct color_ctrl_page *color_ctrl;
const int nr_colors = x; //specify number of colors to be used
nr_pages = //Pages to be allocated
for (i = 0; i < nr_pages; i++) {
    ctrl->colors[i] = color;
    ctrl->pages[i] = 1;
    color = (color + stride) % nr_colors;
}
```

Step 2: In tlb_malloc.c call color_malloc function while initializing the free list.

```
static int init()
{
    memory_req_4k_pages = memory_req_4k_pages + PAGE_SIZE_4K + ( no_of_dtlb_sets_4k * PAGE_SIZE_4K);
    while( pages_added != 32)
    {
        setup_pages(color_ctrl,start++,1,0);
    }

```
va_address =color_malloc_or_exit(4096);

tlb_set_no = get_dtlb_set(va_address);
if(tlb_set_no < 32)
    enum color page_color = get_color(memory_ptr);
    {
        enum color page_color = i;
        struct mblock * block_to_insert = va_address;
        block_to_insert -> length = PAGE_SIZE_4K;
        block_to_insert -> next = NULL;
        insert(small_free_list_ptrs_4k, tlb_set_no, block_to_insert);
        pages_added ++;
    }
else
    {
        printf("\nFailed");
    }
}

Second Approach:

Step 1: Created color control structure by specifying number of pages to allocate and number of colors to be used.

struct color_ctrl_page *color_ctrl;
const int nr_colors = x; //specify number of colors to be used
nr_pages = //Pages to be allocated
for (i = 0; i < nr_pages; i++) {
    ctrl->colors[i] = color;
    ctrl->pages[i] = 1;
    color = (color + stride) % nr_colors;
}

Step 2: Allocate memory of DTLB capacity and then pass virtual address to the mmap depending on the required color.

for(i=0;i<5 ;i++)
{
    setup_pages(color_ctrl,i,1,0);
    struct mblock *ptr = small_free_list_ptrs_4k[i]->next;
    struct mblock *nextptr = ptr->next;
for (j=0; j < dtlb_assoc_4k; j++)
{
    ptr = color_malloc(4096, (void*)ptr);
    ptr->length = PAGE_SIZE_4K;
    ptr->next = nextptr;
    ptr = ptr->next;
    if (ptr != NULL)
        nextptr = ptr->next;
}

Step 3: Modified color_malloc to take an additional parameter i.e. virtual address to add it as an extra
parameter which is passed to mmap function.

Based on this virtual address, it should ideally take it as a hint about where to place the mapping, the
mapping will be created at nearby page boundary. The address of the new mapping is returned as the
result of the call.

static int map_file(const char* filename, void **addr, size_t size, void *virtual_address)
{
    int error = 0;
    int fd;
    int map_failed = MAP_FAILED;
    if (size > 0) {
        fd = open(filename, O_RDWR);
        if (fd >= 0) {
            *addr = mmap(virtual_address, size,
                        PROT_READ | PROT_WRITE,
                        MAP_PRIVATE,
                        fd, 0);
            if (*addr == MAP_FAILED)
                error = -1;
            close(fd);
        } else
            error = fd;
    } else
        *addr = NULL;
    return error;
}

void* color_malloc(size_t size, void *virtual_address)
{
    int err;
    void *mem;
err = set_color_page_info(&color_ctrl);
if (err) {
    fprintf(stderr, "The color page info system call failed.\n");
    mem = NULL;
    goto out;
}

err = map_file(LITMUS_COLOR_ALLOC, &mem, size,virtual_address);
if (err)
    mem = NULL;
out:
    return mem;
}

Problems: litmus_color_alloc_mmap changes the mapping of virtual address and does not return the nearby virtual address making the solution infeasible.

Other Problems faced:

1. Color_malloc fails on more than 32 pages allocation.
2. Tlb_malloc initialization requires 64 pages so could not test it for large number of pages.
3. ColorMalloc documentation
4. No color_free library call to reclaim the colored pages

Bank Coloring:

Problem: In multicore platforms, banks are typically shared among all cores, even though programs running on the cores do not share memory space. In this situation, memory performance is highly unpredictable due to contention in the shared banks.

Memory performance in multicore platforms can vary significantly depending on how data are located in the banks and how the banks are shared among the cores at a given time. Figure 1 shows the best and the worst-case memory access scenarios in multicore processors: when all cores are accessing data located in different memory banks (best-case), requests can be processed in parallel. On the other hand, when all cores are accessing data located in the same memory bank (worst-case) at the same time, requests would be delayed due to contention in the bank.

Solution: Dynamically partition banks to avoid bank sharing among cores, thereby improving isolation on multicore platforms without requiring any special hardware support. Designer can create a virtual scheduling partition for each core and assign private DRAM banks for each partition.

Problems with the solution: Partitioning DRAM banks is not free in the sense that processes in a partition cannot use more memory than the size of the allocated DRAM banks, even though the rest of the DRAM banks are not used.
Fig 8. Memory access scenarios in multicore

Design:

Determined the address mapping for DRAM banks on Tegra3 by using the below specification:

DRAM of Tegra3:

The multi-protocol DDR memory controller can be configured to provide 16-bit or 32-bit-wide accesses to a 1 GB address space using a single rank configuration of 8-bit, 16-bit or 32-bit DRAM memories. ECC is supported in 16-bit bus access mode.

Bank Address Mapping for Tegra3:

The 2 LSB of the linear address are ignored as the address granularity of the DRAM is 32 bits/4 bytes

- Bits[9:2] of the linear address are mapped as column bits[7:0]
- Bits [11:10] are bank bits.
- Bit [12] is a bank bit if the device has more than two bank bits.
- The next bits of the linear address are mapped as column bits, as many as remaining after previous mapping
- The next bits of the linear address are mapped as row bits, as many as needed for the selected device
- The next bit is a device bit if device bits are needed
The number of bank, column, row, and device bits is limited by the number of address pins available:

- Bank width: 2 or 3
- Column width: 8 to 11
- Row width, depends on if you’re using DDR3 with 1 device:
  - Yes: up to 16 bits
  - No: up to 15 bits
- Logical Devices (aka chip-selects): 1 or 2

When two logical devices are used, the total memory mapped by the second device must be less than or equal to the first device. The second device also may have a different row, bank, column mapping from the first device.

Based on the bank bits [10, 11] for 4 banks and [10, 11, 12] for 8 banks we can assign different colors to each bank and control the allocation to avoid interference.

**Conclusion and Future Work:**

Two approaches were used to integrate color cache malloc and tlb malloc:

1. **mmap approach:** To pass an additional parameter of virtual address to color_malloc based on the tlb_set color.
   
   **Problem:** litmus_color_alloc_mmap does not map the virtual address as requested. This can be fixed in future.
2. Allocate single page by using color_malloc, determine color and check the virtual address it is mapping to in tlb_set and populate the free list accordingly for each tlb_set color.

**Problem:**
Cannot allocate more than 31 pages using color_malloc which it should clearly be able to.
No color_free library call in liblitmus to deallocate the colored pages.

---

**References**

- Intel 64 and IA-32 Architecture’s software development manual, Volume 1
- Intel 64 and IA-32 Architecture’s software development manual, Volume 3A
- Intel 64 and IA-32 Architecture’s software development manual, Volume 3B
- Accurate TLB and cache characterization using hardware counters
- [http://www.seco.com/](http://www.seco.com/)
- Technical reference manual Tegra3
- Making Shared Caches More Predictable on Multicore Platforms Bryan C. Ward, Jonathan L. Herman, Christopher J. Kenna, and James H. Anderson, Department of Computer Science, University of North Carolina at Chapel Hill
- PALLOC: DRAM Bank-Aware Memory Allocator for Performance Isolation on Multicore Platforms Heechul Yun, Renato Mancuso, Zheng-Pei Wu and Rodolfo Pellizzoni, RTAS’14