**Virtual Memory System Simulator**

**CYN Dwith (114233325) cyndwith@umd.edu,**

**Koutilya PNVR (114281883), \_\_\_\_\_@umd.edu,**

**Department of Electrical and Computer Engineering,**

**University of Maryland College Park**

**Contents**

1. **Introduction**
   1. **Program Flow**
2. **Motivation and Approach**
   1. **One Level Cache Hierarchy**
3. **Implementation**
   1. **Table Look-aside Buffers (iTLB/dTLB/TLB/VPT)**
   2. **Cache Memory (L1, L2, VL3)**
   3. **Graphical User Interface**
4. **Results and Analysis**
   1. **User Inputs**
   2. **Graphs and Plots**
   3. **Conclusion/Inference**
5. **Future Work**

**Appendix**

1. **C++ Implementation Code**
2. **User Guide for GUI**

**Virtual Memory System Simulator**

**C YN Dwith (114233325)**

**Koutilya PNVR (114281883)**

**Abstract**

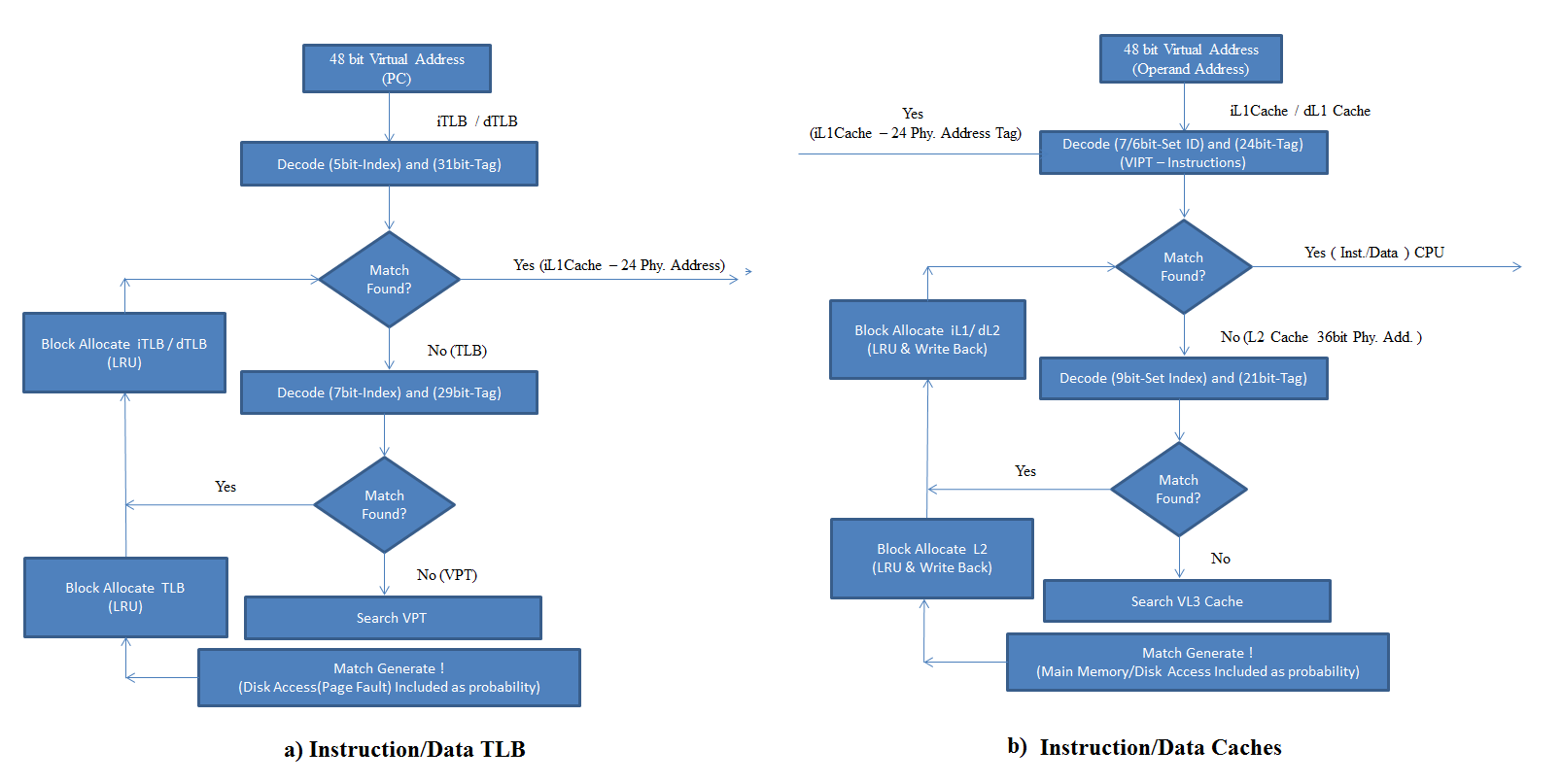
**In this project, we present Virtual Memory Simulator (VMS), to simulate the operations that happen in Intel i7 memory hierarchy. It emulates the time stamping of key steps such as address generation and computation, tag searches in level 1, 2, and 3 caches, level-1 instruction and data table look aside buffers, table look aside buffer, page faults, and virtual to physical address translations. The framework was developed on two programming languages (C++) with emphasis on performance metric like cache misses and page faults. A Qt C++ based GUI is used as visualization tool to graph and compare various performance measures of VMS results such as Miss rate of different caches.**

1. **Introduction**

Simulation tools are used to evaluate performance characteristics of memory system design which are primarily impacted by cache misses and miss rate. The aim of the project is to implement such simulator for modern i7 memory hierarchy and understand the intricate details of key steps such as address translation, Cache searching mechanism, Cache replacement (LRU/pseudo LRU) and write back policy. The performance of the system is measured based on statistical data of clock cycles per Instruction (CPI), miss rate (L1, L2, L3) cache, page faults etc. In this project we try to explore the possibility to improve the simulation results to include spatial and temporal locality which are critical for analysis.

1. **Program Flow**

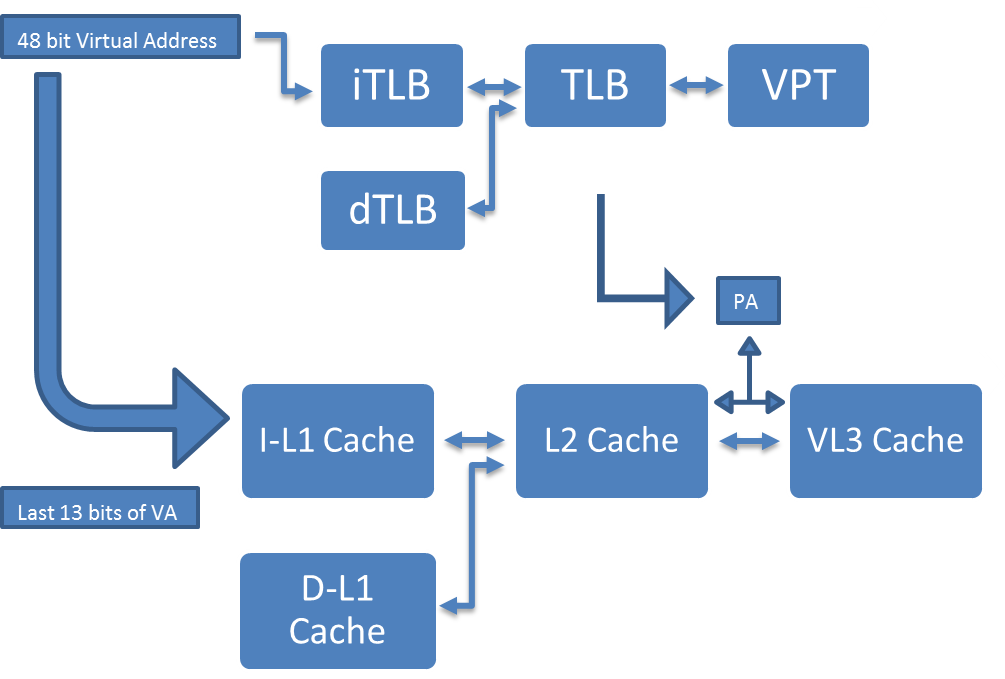
The flow chart below describes the overview of Instruction and Data TLB and Cache access operation, which is covered in detail under Implementation Section.<<Some points on LRU/Cache Inclusivity/Page Coloring>>

****

This report covers the details of implementing Virtual Memory System Simulator and studies the implementation details of three level cache memory system. Project is implemented under certain assumptions for L3 Cache and beyond memory systems to make the implementation plausible and restricted in size, the details of which are covered in the implementation section.

1. **Implementation**

The schematic of the project is shown in the figure. The general implementation is as follows: A 48-bit virtual address is first requested by the CPU which is translated into 24-bit physical address by Table Look-aside Buffers (iTLB/TLB/VPT). The instruction is then fetched into Instruction Register (IR) starting by searching in i-L1, L2 and VL3 caches sequentially by their respective searching mechanisms. After this depending upon the type of instruction and number of operands, the operand address which is 48-bit virtual address again is translated into 24-bit physical address by dTLB/TLB/VPT. Now operands are fetched into data register starting by searching in d-L1, L2 and VL3 caches sequentially by their respective searching mechanisms. The value of the 48-bit PC (Program Counter) is increased by 16 if the executed instruction is not a branch instruction. If it is then PC will be updated by the branch address and the next instruction and its operands are fetched similarly and so on.



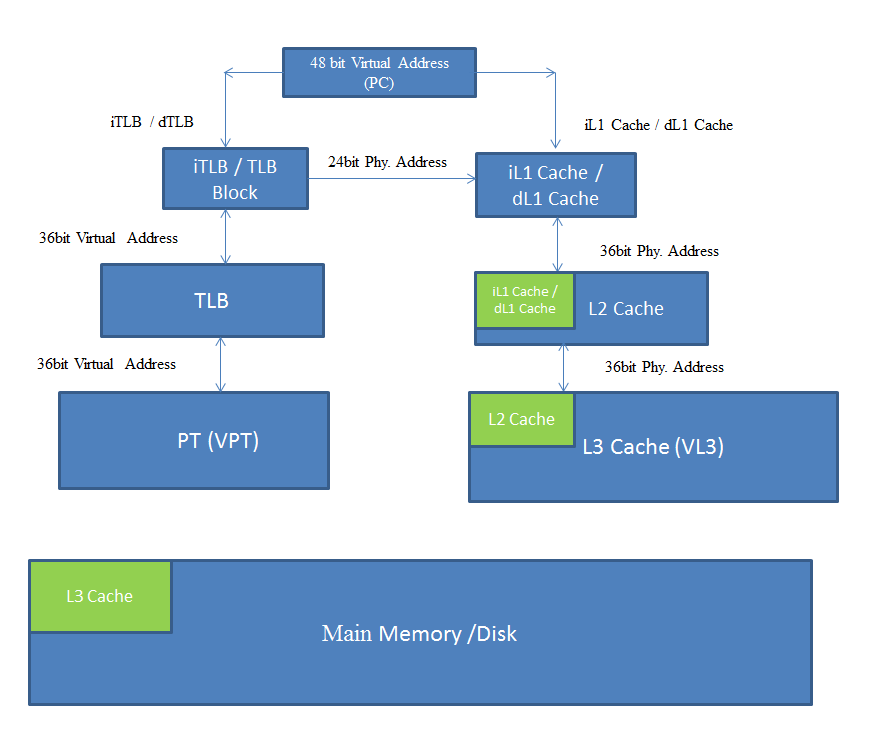
1. **Table Look-aside Buffers (iTLB/dTLB/TLB/VPT)**

The 48-bit virtual address (36+12) instruction (operand) address is translated into 36-bit physical address (24+12) but searching initially in iTLB (dTLB) by its predefined searching mechanism. If it’s a hit i.e., a match is found in iTLB (dTLB) the physical address is returned to memory controller to fetch the instruction (operands) into IR (DR) by searching in L-1, 2 and 3 caches. If no such match is found in iTLB (dTLB) then the search is continued in TLB which is common for both operands and instructions as shown in the figure. If a match is found in TLB, the mapping is written in iTLB at an appropriate location and the physical address is returned to memory controller to search in caches. If no match is found then the search continues in VPT (Virtual Page Table) and the same procedure continues but the implementation part is somewhat different here. After searching in entire VPT, if no match is found, we use a random variable that predicts whether there is a match in VPT or not (because of the large size of PT, we are doing this). If it predicts a match, all the procedure of writing it back in TLB and iTLB (dTLB) is same. But if it predicts no match then we report a page fault and go into the disk to fetch the corresponding page and write it in VPT, TLB and iTLB (dTLB). Nevertheless, after this procedure, we end up with a 36 bit physical address (24+12 bits page offset).

1. **Cache Memory (L1, L2, VL3)**

The search process for i-L1 cache implemented in this project is Virtually Indexed Physically Tagged (VIPT). The index at which the physical address tag needs to be matched in i-L1 is already selected (by using the last bit of Virtual page address tag) by the time we start comparing the tag bits among the 4 blocks from different banks. By this mechanism we can increase the hit time for i-L1 cache. The searching mechanism for the rest of the caches (L2, d-L1) doesn’t follow the same as by that time we would already have the physical address from TLBs. Like in TLBs, the search mechanism looks for a hit in i-L1 (d-L1) cache, once we have the physical address of the instruction (operand). If a hit in i-L1 (d-L1), then the instruction (operand) is fetched into Instruction (Data) Register (IR (DR)). If a miss is registered then the search starts in L2 cache which is common for both i-L1 and d-L1 caches. According to its own searching mechanism, the 36-bit physical address is divided into index and tag fields depending upon the cache design and the searching starts. If a hit is found in L2 then the entire block is written into i-L1 (d-L1) at a specific location decided by the Least Recently Used (LRU) Algorithm.

The algorithm is best suited here as it complements well with the temporal locality property of caches. The least recently used block has lesser probability of being utilized by CPU, hence we can assume it’s safe to replace this block without much affecting the hit time of the cache. After replacement in i-L1 (d-L1) cache, the instruction is now fetched into IR (DR). But, if a miss is registered in L2 cache too, then we start searching in L3 cache. Because of its large size, we implemented a virtual L3 cache just like we did in Virtual Page Table case. After searching through the length of the VL3 cache if we didn’t have hit then we flip a coin again to randomize the possibility of hit in VL3. If we get 0, then we assume there is a hit in VL3 and write the block in L2 and i-L1 (d-L1) cache using LRU algorithm. But, if we get 1 then we report a VL3 miss and start searching in **main memory**. But any memory system here on will only be bigger than L3 cache. Thus we implement the same randomizing mechanism to determine a hit or miss in MM. If a miss, we go to disk, write it in MM, VL3, L2, i-L1 (d-L1) and finally we will fetch the instruction (operand) into IR (DR).



During operand fetching, if the decoded instruction is “Store” then one of the operands need to be stored with a new value i.e., the data in the d-L1 cache (if at all a hit is found) is updated with the new value and the dirty bit field of this block is changed to 1 for this operand address. Doing so will make the memory hierarchy not coherent. To avoid this we follow a process called “write-back” i.e., whenever in future, if the same block is replaced during LRU implementation, we need to carry on this updated data into higher levels (L2, L3 and so on) and store it there. The dirty bit being 1 helps us finding that the data in the block in d-L1 cache is changed but not written back yet into L2 and so on. Doing this will prevent coherency issues.

**What was accomplished?**

1. Evaluation of important parameters like miss rates in every cache and TLB
2. Statistics and time taken at each stage such as address generation and computation, tag searches in level 1, 2, and 3 caches, level-1 instruction and data table look aside buffers, table look aside buffer, page faults, and virtual to physical address translations
3. Graphical User Interface of
   1. How these parameters vary with the number of instructions.
   2. Choosing and entering some options like “no branches”, “Maximum number of branches”, “All are branches” and observing the impact on the above parameters to check whether there is any improvement in the memory system performance.

**What else could have been implemented?**

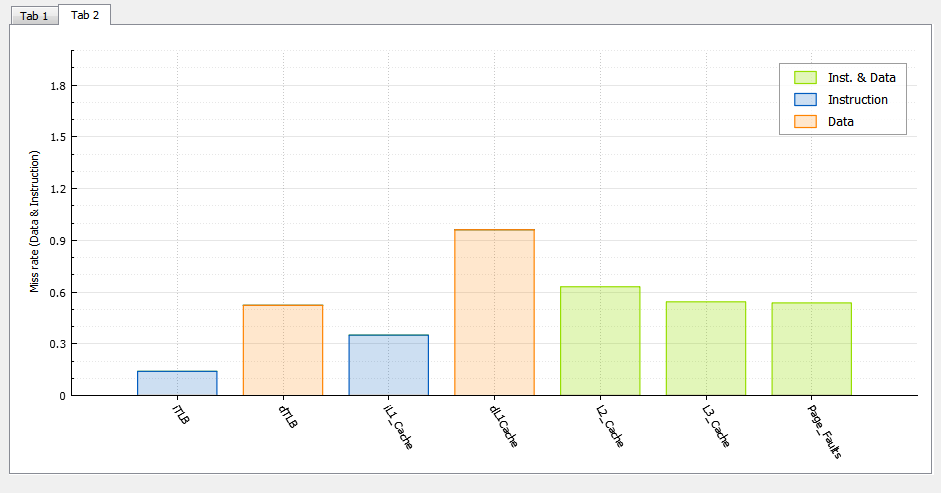
1. File input of instructions: Instead of randomly generating instruction in VL3 cache, we could input a program file that already contains instructions and load them onto VL3, L2 and L1 caches so that the user can witness what type of instructions are being fetched and how often.
2. Realistic probabilities of branching: We could have adopted a way to vary the probability of an instruction fetched to be a “branch”, so that parameters like miss rates can be studied and observed much more effectively
3. One more thing we were interested in but ran out of time was to test an alternate way the “write back” is performed. Instead of writing back to all the lower level caches, main memory and disk (every time a write-back is performed), what if we use the same dirty bit concept in L2 and VL3 caches i.e., writing back to only one lower level cache and making the dirty bit “1” in the new block of the new cache. Whenever this block in the new cache is about to be replaced, the LRU algorithm finds the dirty bit to be 1 and thus “write back” will be performed to the next lower cache of this new cache and so on. This approach saves time as we don’t need to write back the updated value in disk every time a write-back is performed.

**Results and Analysis:**

* + 1. **Graphical User Interface:**

GUI is used as a visualization tool to display all the statistics relevant to the performance of the system. The main parameters displayed are iL1 miss rate, dL1 miss rate, L2 miss rate, VL3 miss rate, iTLB misses, dTLB misses, page faults, CPI etc. by varying the number of instructions fetched with some user input options like “no branches”, “Maximum number of branches” and “All are branches”. These options gives us fluidity to vary them and see the impact on the performance parameters which is key for understanding the memory system.

1. **Parameters : (Discuss a little on the different parameters and their significance)**
   1. **Miss rate of different blocks for (100 Instructions)**



* 1. **Statistics after executions of 100 random Instructions**

In the figure below, we plot Miss rate and (Misses/Instructions) for iL1 Cache, dL1 Cache and L2 Cache. The summary of the all the statistics is present on the right side. The x-axis of the plot is intermediate miss rate after executing 5% of the instructions.

