Inverted Page Tables

(c) 2022 Robert Finch

# Intro

An inverted page table is a table used to store address translations for memory management. The idea behind an inverted page table is that there is a fixed number of pages of memory no matter how it is mapped. It should not be necessary to provide for a map of every possible address, only addresses that correspond to real pages of memory. Each page of memory can be allocated only once. It is either allocated or it is not. Compared to a non-inverted paged memory management system where tables are used to map potentially the entire address space an inverted page table uses less memory. There is typically only a single inverted page table supporting all applications in the system. This is a different approach than a non-inverted page table which may provide separate page tables for each process.

# The Simple Inverted Page Table

The simplest inverted page table contains only a record of the virtual address mapped to the page, and the index into the table is used as the physical page number. A translation can be made by scanning the table for a matching virtual address, then reading off the value of the table index. Unfortunately, the simplest inverted page table is not practical when there are thousands or millions of pages of memory. It simply takes too long to scan the table. The alternative solution to scanning the table is to hash the virtual address to get a table index directly.

# Hashed Table Access

Hashes are great for providing an index value immediately. The issue with hash functions is that they are just a hash. It is possible that two different virtual address will hash to the same value. What is then needed is a way to deal with these hash collisions. Hash collisions can be dealt with using a chain with each link in the chain pointing the to next page table entry to use in the event of a collision. The inverted page table is slightly more complicated then as it needs to store links for hash chains.

# Shared Memory

Another issue to deal with is shared memory. Sometimes applications share memory with other apps for communication purposes, and to conserve memory space where there are common elements. With a paged memory management system, it is easy to share memory, just modify the page table entry to point to the same physical memory as is used by another process. With an inverted page table having only a single entry for each physical page is not sufficient to support shared memory. There needs to be multiple page table entries available for some physical pages but not others because multiple virtual addresses might map to the same physical address. One solution would be to have multiple buckets to store virtual addresses in for each physical address. However, this would waste a lot of memory because much of the time only a single mapped address is needed. There must be a better solution. Rather than reading off the table index as the physical page number, the association of the virtual and physical address can be stored. Since we now need to record the physical address multiple times the simple mechanism of using the table index as the physical page number cannot be used. Instead, the physical page number needs to be stored in the table in addition to the virtual page number.

That means a table larger than the minimum is required. A minimally sized table would contain only one entry for each physical page of memory. So, to allow for shared memory the size of the table is doubled. This smells like a system configuration parameter.

# Thor2022 Inverted Page Table Setup

We have determined that a page table entry needs to store both the physical page number and the virtual page number for the translations. In addition to those two items additional bookkeeping bits are stored. The physical and virtual page numbers are allowed 52 bits each for a total of 104 bits. The other bookkeeping bits require 29 bits of storage, so the total is 133 bits. It is also probably a good idea to leave a little bit of extra space to expand the virtual address for instance. Software often needs to store additional information in the page table entry as well. So, the structure allows for 160 bits.

## Page Table Entry

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 24 | 23 | 22 | 2119 | 1816 | 15 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| ASID | V | G | ~3 | rwx | PL | D | U | S | A | C | R | W | X |
| Physical Page Number31..0 | | | | | | | | | | | | | |
| ~12 | | | | Physical Page Number51..32 | | | | | | | | | |
| Virtual Page Number31..0 | | | | | | | | | | | | | |
| ~12 | | | | Virtual Page Number51..32 | | | | | | | | | |

Fields Description

|  |  |
| --- | --- |
| ASID | Address Space Identifier |
| V | translation Valid |
| G | Global translation |
| PL | Privilege Level needed to access the page |
| D | dirty |
| U | unspecified use |
| S | Page Size |
| A | Accessed |
| C | Cacheable |
| R | Readable |
| W | Writeable |
| X | Executable |
| rwx | system mode R,W,X |

The page table does not include everything needed to manage pages of memory. There is additional information such as share counts to take care of, but this information is better managed in a separate table.

## Page Table Groups

We want the search for translations to be fast. That means being able to search in parallel. So, PTEs are stored in groups that are searched in parallel for translations. Access to the group should be as fast as possible. There are also hardware limits to how many entries can be searched at once while retaining a high clock rate. So, the convenient size of 1024 bits was chosen as the amount of memory to fetch. This corresponds to two cache lines worth of data, and eight 128-bit burst accesses to the memory controller.

A page table group then contains six page-table entries. All entries in the group are searched in parallel for a match.

|  |  |
| --- | --- |
| 160 0 | |
|  | Group Header |
| PTE0 | |
| PTE1 | |
| PTE2 | |
| PTE3 | |
| PTE4 | |
| PTE5 | |

For a system with 512MB of RAM:

There are 131072 4kB pages. 131072 PTEs are required without considering sharing. Since we decided to double the table size for sharing 262144 PTEs need to be accommodated. This works out to 43691 groups of PTEs, or about 5.6 MB. 43691 is an ugly number. Let us provide some extra groups and make it an even 65536. To get to a page table group fast a hash function is needed then that returns a 16-bit number.

## Hash Function

While the asid should be considered part of the virtual address and could be used as part of the hash there is another smaller number, the app id, that serves a similar purpose. The hash function chosen uses the app id combined with virtual address bits 14 to 25. This should space out the PTEs according to the app. Address bits 14 to 25 where chosen, ignoring higher order address bits because most code modules are less than 16MB in size. Hence the virtual address is often less than 16MB. Address bits 0 to 11 are the page offset and do not need to come into the calculation. Address bits 12 and 13 select one of four address ranges. the PTG supports six PTEs. The translations where address bits 12 and 13 are involved are likely consecutive pages that would show up in the same PTG.

## Finding a Match

Once the PTG to be searched is located using the hash function, which PTE to use needs to be sorted out. The match operation must include both the virtual address bits and the asid, address space identifier, as part of the test for a match. It is possible that the same virtual address is used by two or more different address spaces, which is why it needs to be in the match.

## Page Table Group Header

The page table group header contains a link to the next page table group to search in the event of a hash collision, when there is not enough room in the current PTG.

## Locality of Reference

The page table group may be cached in the system read cache for performance. It is likely that the same PTG group will be used multiple times due to the locality of reference exhibited by running software.

# The TLB – Translation Lookaside Buffer

If every memory access required two or three additional accesses to map the address to a final target access, memory access would be quite slow, slowed down by a factor or two or three, possibly more. To improve performance, the memory mapping translations are stored in another unit call the TLB standing for Translation Lookaside Buffer.