Prospects for Functional Address Translation

Conor Hetland  Georgios Tziantzioulis  Brian Suchy
Kyle Hale  Nikos Hardavellas  Peter Dinda

Prescience Lab
Computer Science
Northwestern University

pdinda.org
presciencelab.org
Paper in a Nutshell

• Page tables implement address translation *functions*
• Can we implement these *functions* better on future (malleable) hardware?
  – Faster/more space efficient/easier to generate+update
• We studied four models for doing this
  – How well do they work for actual x64 page tables from HPC and other workloads?
• Results are mixed
  – Most promising technique: multiplexor tree
Outline

• Address translation functions
• Traditional paging and alternative models
• Our techniques and results
  – Perfect hashing for inverted page tables
  – Espresso-minimized PLAs
  – Task-specific functional language for FPGA synthesis
  – Multiplexor trees
• Conclusions and future work
  – CARAT
Motivation

• Address translation is a hot topic again
• Driven by... changing workloads
  – Database engines, cloud services, unikernels
• Driven by... TLB costs
  – TLB misses => lower performance
  – TLB reach may be insufficient
  – TLBs consume significant power/energy/area
  – Virtualization compounds these

• Our interest: parallel workloads and HPC
  – Part of a project to rethink the parallel hardware/software stack (interweaving.org)
Address Translation Functions

**Every** Address (ifetch, data read/write/ control access / etc)

- Virtual Page Number (VPN) and Offset
- Physical Page Number (PPN) and Offset
- Metadata: Protection, mode, cacheability, etc

Radix tree implements this function

- x64: 4-level radix tree, 512 entries per node
- Worst case: 5 memory accesses
- with VMM: 25 memory accesses
Traditional Paging (x64, others similar)

- TLB essential
  - Must avoid almost all actual page table references
- TLB design and cache design are tightly coupled, limiting cache design
It Doesn’t Need to be Page Tables

Every Address (ifetch, data read/write/ control access / etc)

Any representation of the function could work

Virtual Page Number (VPN)  Offset
Inverted page tables
1 step access without hash collisions
Bits to bits => Truth Table
Reconfigurable PLA-like logic
Multi-level Logic
Reconfigurable FPGA-like logic
Run-length-encoding with hardware search tree
FPGA or direct implementation

Metadata  Physical Page Number (PPN)  Offset
Protection, mode, cacheability, etc
Locales for Functional Address Translation (FAT)

Most Restrictive - Lowest Latency / Smallest State

Integrate with Existing Cache Structures

Integrate with Existing Cache Structures

Change Cache Structures

Core-TLB

Core-Pagewalker

Edge-TLB

Edge-Pagewalker

Least Restrictive - Highest Latency / Largest State
Core-TLB

• Most extreme

• Replace TLB!
  – Perhaps without misses!

• Must operate at least as fast as a TLB
  – Cannot have much state
Workloads

Our tool captures Linux page table snapshots (processes)

• General Purpose Servers
  – 2 machines, 19 days, ~1.2 million snapshots
• Mantevo
  – 13 snapshots, mid-run, of 13 benchmarks
• NAS
  – 40 snapshots, 4 each over 10 benchmarks
• PARSEC
  – 70 snapshots, 4-14 each over 7 benchmarks
• HPCCG on Legion
  – 221 snapshots, 1 second intervals
• Synthetic
  – 3 contrived snapshots
Evaluation Criteria

• Generation time
  – How quickly can we transform the mapping into the function?
    • By direct measurement of cost of software tool

• Space complexity
  – Estimate hardware resource cost
    • Ideally, by synthesis to FPGA via Verilog using Quartus, then counting logic blocks

• Lookup time
  – Estimate cost of lookup
    • Ideally by path length / cycle time of FPGA

• Caveat: update time not considered
Inverted Page Tables with Perfect Hashing

- IPT provides single-level page table entry lookup without hash collisions...

- Perfect hashing makes hash functions that have no collisions

- Two generators considered
  - Minimum Perfect Hashing – CHM algorithm from CMPH
  - Perfect Hashing – GPERF

- CHM/CMPH is most interesting
Fig. 5. Minimum perfect hash function generation time using CMPH versus number of page table entries in snapshot.

B. Study

We attempted to construct perfect hash functions for every page table described in Figure 3. This was done sequentially, using only a single logical CPU / hardware thread with no competing workloads. In each case we measured:

- **Generation time:** the time (sys + user) to generate the perfect hash function
- **Space complexity:** The size of the hash function, combining its table space and a proxy (code size) for its likely cost in hardware.

We also analyzed the following by examining the generated functions and table sizes:

- **Lookup time:** The likely cost of a lookup given a hardware implementation of the perfect hash function.

We had no trouble using CMPH on every page table. GPERF is much slower, so we computed hash functions for samples of the Murphy, Hanlon, Mantevo, PARSEC, Legion, and Synthetic page tables. We were unable to generate GPERF perfect hash functions for any of the NAS page tables even after letting the tool run for several days per page table.

Fig. 6. Minimum perfect hash function space complexity versus the number of page table entries in snapshot. Only the Murphy dataset is shown. All others have the identical straight-line behavior.

Fig. 7. Non-minimum perfect hash function space complexity versus the number of page table entries in snapshot using GPERF. Mantevo and NAS did not complete.

C. Observations

Figures 5 illustrates the measured generation time using CMPH. CMPH works in time proportional to the number of page table entries, as promised theoretically, although there is considerable variation. We do not show results for GPERF here—it is many orders of magnitude slower, and, in fact, the generation time for GPERF, for large virtual address spaces, is prohibitive. Note that we are not casting aspersions on GPERF here—we are in fact asking it to do something (handle a vast keyspace) that it is not designed for.

Figure 6 shows the space complexity measurements for the Murphy dataset using CMPH. Note that this cost is not just theoretically proportional to the number of keys (VPNs), but is also empirically exactly the case for this application. We exclude the other graphs as they look virtually identical in this regard.

Figure 7 shows the space costs for GPERF as a function of the number of PTEs. Mantevo and NAS are omitted since the generation time is impossibly long. The behavior is roughly linear, although not as straightforward as with CMPH. Note
Function Body Is Compact, But Needs Considerable State

- Murphy (a)
- Hanlon (b)
- Mantevo (c)
- NAS (d)
- PARSEC (e)
- Legion (f)

Fig. 5. Minimum perfect hash function generation time using CMPH versus number of page table entries in snapshot.

B. Study

We attempted to construct perfect hash functions for every page table described in Figure 3. This was done sequentially, using only a single logical CPU / hardware thread with no competing workloads. In each case we measured:

- Generation time: the time (sys + user) to generate the perfect hash function
- Space complexity: The size of the hash function, combining its table space and a proxy (code size) for its likely cost in hardware.

We also analyzed the following by examining the generated functions and table sizes:

- Lookup time: The likely cost of a lookup given a hardware implementation of the perfect hash function.

We had no trouble using CMPH on every page table. GPERF is much slower, so we computed hash functions for samples of the Murphy, Hanlon, Mantevo, PARSEC, Legion, and Synthetic page tables. We were unable to generate GPERF perfect hash functions for any of the NAS page tables even after letting the tool run for several days per page table.

Fig. 6. Minimum perfect hash function space complexity versus the number of page table entries in snapshot. Only the Murphy dataset is shown. All others have the identical straight-line behavior.

Fig. 7. Non-minimum perfect hash function space complexity versus the number of page table entries in snapshot using GPERF. Mantevo and NAS did not complete.

C. Observations

Figures 5 illustrates the measured generation time using CMPH. CMPH works in time proportional to the number of page table entries, as promised theoretically, although there is considerable variation. We do not show results for GPERF here—it is many orders of magnitude slower, and, in fact, the generation time for GPERF, for large virtual address spaces, is prohibitive. Note that we are not casting aspersions on GPERF here—we are in fact asking it to do something (handle a vast keyspace) that it is not designed for.

Figure 6 shows the space complexity measurements for the Murphy dataset using CMPH. Note that this cost is not just theoretically proportional to the number of keys (VPNs), but is also empirically exactly the case for this application. We exclude the other graphs as they look virtually identical in this regard.

Figure 7 shows the space costs for GPERF as a function of the number of PTEs. Mantevo and NAS are omitted since the generation time is impossibly long. The behavior is roughly linear, although not as straightforward as with CMPH. Note
Lookups Are Tricky

- Hash function involves two memory references
  - With Inverse PT references, close to a forward PT...

- GPERF has contrasting results
  - MUCH slower to generate – no O(n) guarantee
  - Function body larger, but state much smaller
  - Lookup likely quite fast since state will not require memory accesses
Espresso-Minimized PLAs

- Address translation function as a bit-by-bit truth table

- Classic Berkeley Espresso truth table minimizer for PLA-style reconfigurable logic
  - Imagine future processor with a PLA, not an FPGA

- For space complexity: synthesis by Quartus for FPGA
  - Limitation of evaluation
  - Real hardware would take Espresso output directly
Generation Costs Are High

• 10s of minutes of CPU time for Espresso-minimization
  – Would always be needed

• Hours to synthesize into FPGA logic
  – Only needed for our evaluation approach

• Generation time limits the number of snapshots we can consider
Space Complexity Is Probably Linear in Address Space Size – Plus No Extra State
Lookup Time Likely Fast

• PLA is fundamentally two-level logic
• Provided sufficiently high fan-in/fan-out, lookup would be single-cycle
• Lots of caveats here...
Motivating Multiplexor Trees: Page Table Entries Respond Well to Run-length-encoding

- Contigified Regions
- Original Regions
- PTEs

*Figure 12. CDFs of address space sizes as measured by PTEs (rightmost curves)*
Multiplexor Trees

• Think Linux memory map red-black tree...

(VPN, length) -> PPN

• ... but realized as a hardware decision tree
A Two-Region Multiplexor Tree in Verilog

```verilog
always @(vpn)
if ((vpn>=reg_vpn[0]) &&
    (vpn<(reg_vpn[0] + reg_num[0])))
    out = reg_ppn[0] + (vpn - reg_vpn[0]);
else
    if ((vpn>=reg_vpn[1]) &&
        (vpn<(reg_vpn[1] + reg_num[1])))
        out = reg_ppn[1] + (vpn - reg_vpn[1]);
    else
        out = 'hfffffffffffffff;
```

• Search process is this hardware tree
A 16-Region Multiplexor Tree in Verilog (trust me!)
Bespoke Multiplexor Trees

- Generate *stateless* search tree *specific* to the run-length encoded address translation function
Registered Multiplexor Trees

• Generate *generic* multiplexor tree of n regions that can be *loaded* with any run-length-encoded function that has n or fewer regions.
Generation Costs

• Generally high (hours)
  – Milliseconds to generate, hours to synthesize

• Matters for bespoke

• Does not matter for registered
  – We can reuse the synthesized hardware just by loading its registers with other values
Space Complexity is Reasonable (Bespoke)

Logic Elements versus Contiguous Regions

Total Logic Elements (Cyclone IV GX)

Number of Contiguous Regions
All Mappings Transform Mapping into Regions Logarithmic Lookup Generation Time

Space Complexity is Reasonable (Registered)

Fig. 13. Address translation by bespoke multiplexor tree.

We consider two approaches to producing a multiplexor tree. In a bespoke multiplexor tree, illustrated in Figure 13, all VPN mappings are known at synthesis time. Because of this, all regions are constants from the perspective of synthesis. We have developed a tool that first transforms VPN mappings into region mappings. The tool next generates the specific search tree needed for those regions, and then produces a Verilog version of the tree as combinational logic. Finally, the Verilog is synthesized into an FPGA (using Quartus 17) in our implementation.

Figure 14 shows the process for producing a registered multiplexor tree. Here, the regions are not known a priori. Instead, given a bound on the number of supported regions, a separate tool produces a multiplexor tree in Verilog whose regions and region-splitting VPNs are not constants, but registered values. That is, it produces a multiplexor tree that is parameterized by runtime information. The tool also produces the I/O logic needed to dynamically configure the registers, and a software interface that maps regions to the appropriate registers, and determines and loads the splitting VPNs. The multiplexor tree is synthesized only once. Any address translation function can then be loaded into it via the software interface at the cost of loading a number of registers proportional to the number of regions.

Our target was the same FPGA as described in Section V. We measured both processes in the following ways:

- **Generation time**: The time to generate the Verilog from input regions (bespoke) or the number of regions (registered), plus the time to synthesize the FPGA block.
- **Space complexity**: The size of the resulting FPGA block in terms of the FPGA's native logic elements.

In terms of space complexity, we would expect that any multiplexor tree to take $O(n)$ for $n$ regions. Figures 15 and 16 show the empirical space complexity as a function of the number of regions for 512 to 1024 regions. The measured space costs clearly follow the expected linear behavior.

The worse case of the parallel lookup through the tree is $O(\log n)$. Given how compactly real address translation functions can be represented as regions (Figure 12), $n \ll 10^3 \ldots 10^4$ can cover a wide range of cases, particularly...
Lookup Cost

• Log depth circuit (n=number of regions)
• Single cycle for what we synthesized
  – But this is for a slow FPGA
  – And depends on scale
• Bespoke versus registered does not matter

• Kernel can potentially control these costs through memory allocation model
  – Enhance virtual->physical contiguity -> ”contigify”
**Speculation About Best Fits**

*Most Restrictive - Lowest Latency / Smallest State*

- Integrate with Existing Cache Structures
  - Core-TLB: Multiplexor Trees (Registered)
  - Core-Pagewalker: Perfect Hashing/IPT
- Change Cache Structures
  - Edge-TLB: Espresso/PLA
  - Edge-Pagewalker: Perfect Hashing/IPT

*Least Restrictive - Highest Latency / Largest State*
Related Work

• DIY Address Translation (Alam et al, ISCA 17)
• Hash, Don’t Cache (Yaniv et al, SIGMETRICS 16)
• Translation Caching/SpecTLB (Barr et al, ISCA 10)
• Segments / RLE tables (Basu et al, ISCA 13)
• Nested Paging (Bhargava et al, ASPLOS 08)
• Bhattacharjee et al (numerous, present)
• Numerous ideas from the last time address translation was a hot topic (mid ‘90s)
Current Direction: CARAT

• Compiler- and Runtime-based Address Translation

• *No* address translation – all *physical* addressing
  – Optionally – no page abstraction

• Compiler and runtime jointly enforce protection and enable data mobility

• Paper forthcoming!
Paper in a Nutshell

• Page tables implement address translation *functions*
• Can we implement these *functions* better on future (malleable) hardware?
  – Faster/more space efficient/easier to generate+update
• We studied four models for doing this
  – How well do they work for actual x64 page tables from HPC and other workloads?
• Results are mixed
  – Most promising technique: multiplexor tree
For More Information

• Peter Dinda
  – pdinda@northwestern.edu
  – http://pdinda.org

• Interweaving Project
  – http://interweaving.org

• Prescience Lab
  – http://presciencelab.org

• Acknowledgements
  – NSF, DOE, Intel