Prospects for Functional Address Translation

Conor Hetland[*] Georgios Tziantzioulis[†] Brian Suchy[*]
Kyle Hale[‡] Nikos Hardavellas[*] Peter Dinda[*]

[*]Northwestern University [†]Princeton University [‡]Illinois Institute of Technology

***Abstract*—Address translation fundamentally embodies a translation function that maps from virtual to physical addresses. In current systems, the translation function is encoded by the kernel in an in-memory radix tree structure (the page table hierarchy) which is then interpreted by the hardware (the pagewalker, pagewalk-caches, and TLBs). We consider implementing the *translation function itself* as reconfigurable hardware—does this make any sense? To study this question, we collected numerous in-situ Linux page tables for a wide range of workloads, including those from HPC, to serve as example translation functions. We then prototyped several potential mechanisms to implement the translation function, including inverted page tables with function-specific perfect hashing, translation functions directly implemented using Espresso-minimized PLAs, translation functions genetically-evolved in a language suitable for FPGA-like synthesis, and translation functions based on recovered/manufactured region (segment/mmap) lookup using multiplexor trees. Each mechanism was then evaluated using the Linux page tables, primarily for space and lookup speed. We report our findings and try to address the question.**

***Index Terms*—address translation, paging, segmentation, operating systems, high performance computing**

I. Introduction

Address translation was once considered a solved problem, and there was a general agreement in the community that paging was the solution. Now, however, the address translation problem is being revisited extensively, prompted by numerous changes. First, machines are dramatically scaling in terms of physical memory and core count. Second, workloads are shifting from the historic norms on which traditional paging designs were based. Third, power and energy have become first-order concerns, and the hardware that supports traditional paging is not cheap in these regards. Finally, innovation in operating systems, runtimes, and compilers is making it possible to revisit address translation from the software perspective.

Different from related work (Section IX), our own interest in this problem is at the intersection of high performance and parallel computing, operating systems, and hardware/software codesign. We previously developed the hybrid run-time (HRT) model, in which we fuse the parallel application, language run-time, and kernel into a single entity [31]. It is salient that in this model there is only a single address space. In evaluating this model, we developed a kernel framework, Nautilus [30], that implements the address space using identity-mapped paging with the maximum possible page size. This allows for considerable speedups over traditionally paged execution because very few translation lookaside buffer (TLB) misses can possibly occur. In effect, this is the closest thing to turning paging off altogether (which is not currently possible on our target platform, x64).

Of course, discarding address translation entirely poses its own issues. It is not feasible in general purpose computing, and only partially feasible in parallel computing (our above noted work and [32] makes that argument). However, turning paging off lies at one end of a spectrum, with current paging / address translation mechanisms (summarized in Section II) at the other end. Alternative address translation approaches lie in between. Our motivating question is: What is the appropriate alternative address translation mechanism for high performance and parallel computing?

Here we consider one alternative, *functional address translation (FAT)*. The key idea in FAT, on which Section II elaborates, is to treat the mapping between virtual and physical addresses as a function, and to then encode this function *directly*. This is substantially different from current systems, in which the function operates at the granularity of pages, and is encoded in a page table hierarchy, a radix-tree search structure that is aggressively cached by TLBs and other hardware. As far as we are aware, functional address translation has not previously been explored, with DIY address translation [1] and region-based translation [7], [29] being closest.

Given hardware innovation that is placing reconfigurable logic ever closer to the CPU core, there is reason to believe that functional address translation could be integrated into a processor in different ways. A very high speed functional address translation unit could augment the TLB or pagewalker in current systems that use physically tagged caches. Future systems using virtually tagged caches [8], may tolerate a slower functional address translation unit.

What are the prospects for functional address translation? To address this question, we developed several different FAT approaches that differ in how the function is encoded. We evaluated these approaches in the following ways:

1 How long does it take to construct the functional representation? (Generation time.)
2 How much space (hardware resources) does the the functional representation require? (Space complexity.)
3 How long does a lookup take? (Lookup time.)

Our approach to these questions is empirical. We collected a wide range of page table snapshots on current systems running

This project is made possible by support from the United States National Science Foundation through grants CCF-1533560, CNS-1763743, CNS-1763612, and by equipment support from Intel Corporation.

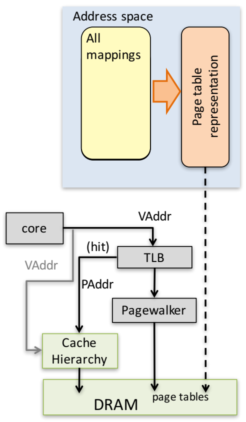


Fig. 1. Traditional address translation.

general purpose and HPC/parallel applications. We attempt to encode each of these snapshots as a function, and directly measure the results.

An additional measure of less importance to us is the cost of an incremental update, although we do consider it. Incremental update is of course something that the traditional address translation approach excels at and FAT approaches likely will not. Recall that we are specifically interested in high performance and parallel workloads. These tend to have quite stable mappings over time, hence incremental updates are less needed. If memory pressure from other processes forces the kernel to change mappings, that memory pressure is likely to wreak havoc on performance anyway. In other words, a parallel application needs to have enough memory, and when this is the case, incremental updates are rare.

Our contributions are as follows:

- We introduce the concept of functional address translation (FAT) and define four models in which it could be integrated into current and future systems.
- We describe four FAT approaches (Perfect Hashing, Espresso-minimized PLAs, Functional Language/Genetic Programming, and Multiplexor Trees).
- We empirically evaluate these approaches by attempting to use them to represent a wide range of page tables captured from existing systems running general and HPC/parallel workloads. Our evaluation focuses on construction cost, space requirements, and lookup speed.
- We identify which FAT approach(es) are most suitable for each of the integration models.

II. Address translation functions

Figure 1 illustrates how address translation via paging operates on most current processors. Our focus is on Intel/AMD x86 processors, so when we give specifics, it is with regard to these processors, in particular when operating in 64-bit mode (i.e., the common "x64" or "x86_64").

Every memory reference, including an instruction fetch, has its virtual address (*VAddr*) translated to a physical address (*PAddr*) that is ultimately the address the memory system uses. Virtualization adds complexity to this address translation process, but is essentially orthogonal to the idea this paper explores.[1] On x64 and other processors, the VAddr is also typically used to immediately begin the data lookup process in the highest level(s) of the cache hierarchy, but before any data is returned, the PAddr resulting from translation is used to determine whether the data found actually belongs to the current address space. Typically, the L1 cache is virtually indexed, but physically tagged. This optimization sets the stage for different ways to potentially integrate functional address translation, which we will describe shortly.

In a paging system, address translation happens at page granularity, so instead of translating VAddr$\rightarrow$PAddr, only the bits of the VAddr that contain the virtual page number (*VPN*) are translated to (and replaced by) bits that contain the physical page number (*PPN*): VPN$\rightarrow$PPN. Strictly speaking, the translation is VPN$\rightarrow$PTE, where the *PTE* (page table entry) contains both the PPN and the access permissions and other metadata about the VPN and PPN. In this paper, we focus on VPN$\rightarrow$PPN.

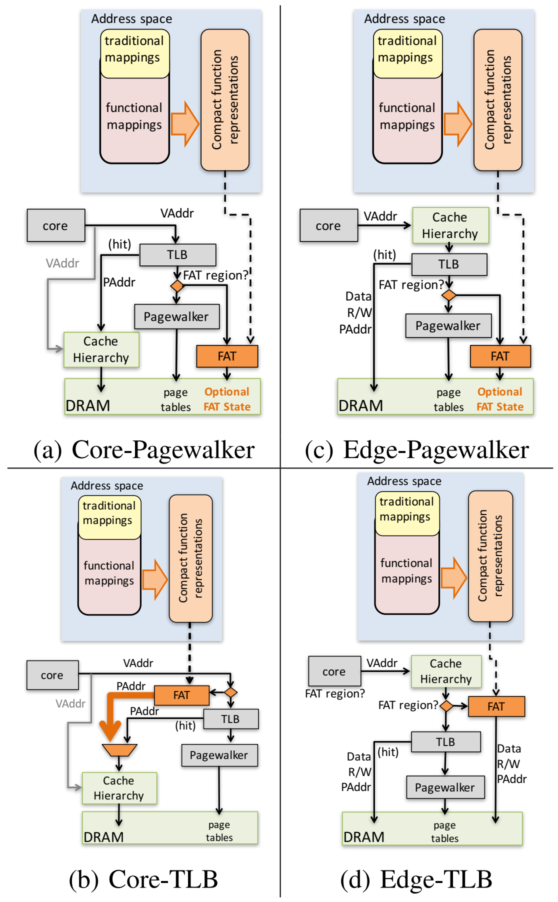
VPN$\rightarrow$PPN is a function whose domain is all currently occupied virtual pages in the current virtual address space, and whose range is all occupied physical pages in the physical address space. This function is determined by the operating system kernel, and its contents may change over time. In current systems the representation of the function is as radix trees that are stored in physical memory. For x64 in particular, this arrangement is a 4-level hierarchy with each level corresponding to 9 bits of the virtual address. The hierarchy allows short-circuiting during traversal to create composite pages (a page can consist of a single base page (4KB in size), 512 base pages (a "large page"), 512^2 base pages (a "huge page"), and (eventually) 512^3 base pages. Intel and AMD do not currently use the entire 64 bit virtual address space, resulting in an apparent mismatch ($9+9+9+9+12 \neq 64$), but there is a model for expanding it by adding more levels over time [37].

There are two central actors in address translation: the translation lookaside buffer (TLB) and the pagewalker.[2] The TLB caches translations that have been read by traversing the in-memory page tables, and its extremely high hit rate is essential to providing modern address translation with little to no extra performance overhead compared to using physical addressing. When the TLB misses, it invokes the pagewalker, which traverses the in-memory page tables using physical addresses until it finds the relevant PTE and places the mapping into the TLB.

TLBs require substantial chip area, energy, and power in order to achieve their necessary extremely low latency [8], [24], [25], [43]. Furthermore, a range of important workloads have been found to have high TLB miss rates, which has spawned a range of research on alternative address translation approaches, which we elaborate on in Section IX. While

[1]Virtualized address translation (shadow and nested paging) could also leverage the idea. See our prior work [2], [35] for more information. The first of these papers introduces inverted page tables for this purposes, and mentions, but does not evaluate the use of perfect hashing in this context.

[2]We include here other elements such as the partial pagewalk caches.



(a) Core-Pagewalker (c) Edge-Pagewalker (b) Core-TLB (d) Edge-TLB

Fig. 2. Possibly models for incorporating functional address translation (FAT).

pagewalkers have more relaxed latency requirements, their speed is critical for properly handling such workloads that do not have good enough spatial and temporal locality and thus have high TLB miss rates. Our work is in particular driven by the address translation demands of parallel and high-performance applications (which display these characteristics).

Since VPN→PPN is a function, a natural question is whether a representation of the function other than via radix trees would lead to faster, smaller, or less power-hungry address translation. The bulk of this paper address this question by considering other representations, particularly those that would be suitable for reconfigurable hardware of some form in the processor. Imagine that we can replace some or all of the address translation ultimately encoded via the page tables with a function that we load into the hardware, the functional address translator (*FAT*). What form should the FAT take?

This question depends on where the FAT is introduced. We think of the FAT as augmenting the traditional address translation path (which could be shrunk as more and more translation is offloaded to the FAT). The traditional path's TLB and pagewalker remain for use with legacy OS kernels and for FAT-aware kernels to use for translations that do not conform well with the particular scheme implemented by the FAT. As with various related works, we assign dynamically selectable chunks of the virtual address space to be translated by the FAT (the *FAT regions*), while any address that is not in a FAT region is translated by the traditional path. The FAT regions might correspond to special `mmap` regions on Unix-like kernels.

Figure 2 illustrates four possible models by which to introduce the FAT. We analyze our FAT representations with respect to these models. In all the models, the kernel is responsible for loading the FAT with the appropriate function for the current address space, as well as modifying or reloading the function as it changes the mapping for any FAT region. The left column of the figure represents models that do not fundamentally change the relationship of translation with the cache hierarchy. In Core-Pagewalker (Figure 2(a)), the FAT is an alternative to the pagewalker and is invoked on a TLB miss in a FAT region. In this model, the FAT may have optional state stored in physical memory and it must operate at pagewalker latencies or faster. In Core-TLB (Figure 2(b)), the most aggressive of our models, the FAT instead forms an alternative to the TLB. Virtual addresses in a FAT region are routed to the FAT *instead of* to the TLB. Here, all state must reside in the FAT, and the FAT must operate at TLB latencies or faster.

The right column of Figure 2 presents models in which we change the relationship of address translation and caching. Here, we assume a virtually indexed and virtually tagged cache hierarchy.[3] As a consequence the TLB (and page walker) can be moved to the "lower edge" of the cache hierarchy. A translation only occurs when a line is brought into the cache hierarchy from DRAM, instead of on every memory reference. As a consequence, the TLB (and pagewalker) have much less stringent latency requirements and also have a much lower throughput because each translation is amortized over the lifetime of the cache line in the cache hierarchy. In the Edge-Pagewalker model (Figure 2(c)), the FAT is an alternative to the pagewalker that is invoked for any FAT region. In the Edge-TLB model (Figure 2(d)), the FAT becomes an alternative to the TLB.

Non-processor models: Address translation occurs outside of the processor in modern machines. For example, the IOMMU allows address translation to be applied to I/O device DMA. Current IOMMUs use paging with similar or identical page tables as described above. Paging in IOMMUs could be augmented or even replaced with FAT.

Another example is Intel's HARP platform, which integrates a processor and FPGA within the same socket [16], or package [52]. HARP provides fast memory system-based communication, with coherence, between the processor and FPGA. The programming environment provides shim layers of logic for the FPGA side that augment this basic coherent memory model. One layer provides the ability for the FPGA side to use virtual addresses, thus allowing the developer's FPGA hardware (the application functional unit (AFU)) to operate within the same address space as the process that it is augmenting. The current implementation of this shim layer is as a TLB with a pagewalker, similar to that of Figure 1. Given that the FPGA is completely malleable hardware, this shim could readily incorporate an FAT.

[3]The issues raised for shared pages and other corner cases are ignored here.

III. Page table snapshots

To assess various approaches to address translation, we collected snapshots of Linux page tables from a range of environments. Our trace data is summarized in Figure 3. The snapshots were collected using a user-level tool described elsewhere [22]. That paper also describes the first two datasets in more detail.

The *Murphy* dataset was collected on a Dell R410 server equipped with 128 GB of memory. It runs Red Hat 6.7 (stock Red Hat-provided 2.6.32 kernel) and Oracle 11g Enterprise 11.2, as well as Apache and other tools needed to build Oracle-based web applications. During the time of the study it was being used to teach a databases course in which 50 students were simultaneously developing applications based on running analysis queries on FEC political contribution data. Over a period of 19 days, at 15 minute intervals, we collected the page table of every process on the machine

The *Hanlon* dataset was collected on a Dell T620 server equipped with 128 GB of memory, and NVIDIA K20 and Intel Phi co-processors. It runs Red Hat 6.7 (stock Red Hat-provided 2.6.32 kernel) and the toolchains needed to support the coprocessors. During the study, it was extensively used in an introductory computer systems course by about 150 students. Over a period of 19 days, at 15 minute intervals, we collected the page table of every process on the machine

Our remaining datasets were collected on a Dell T620 server with 32 GB of RAM, that ran Red Hat 6.5 (stock Red Hat-provided 2.6.32 kernel). These datasets contain page tables from high performance computing and parallel computing benchmark applications and suites.

The *Mantevo* dataset captures the Mantevo benchmark suite [6], [33], which is a collection of "miniapps" that are used by Sandia National Labs, other DOE sites, and the U.S. Exascale Computing Project. A miniapp is an application that has been shrunk to its essential elements in order to make it easier to bring up on a new platform for evaluation, as well as to support hardware/software codesign. We snapshotted each miniapp in mid-run, when its page table had grown to have the most active PTEs. The specific miniapps we measured were: CloverLeaf (hydrodynamics in 2D), CloverLeaf3D (hydrodynamics in 3D), CoMD (molecular dynamics), HPCG (conjugate gradient), MiniAero (computational fluid dynamics), MiniAMR (adaptive mesh refinement), MiniFE (finite element methods), MiniGhost (3D stencil), MiniMD (molecular dynamics), MiniSMAC2D (turbulent fluid flow), MiniXyce (analog circuit simulation), Pathfinder (graph search), and TeaLeaf (heat conduction).

The *NAS* dataset captures the NAS 3.3.1 benchmarks in their OpenMP implementation [41]. The largest problem class that would fit in physical memory was used in each case (i.e., class D, except DC which used class B). Page table snapshots were taken at four points during execution at a 10 seconds interval. The specific benchmarks used were: BT (block tridiagonal solver), CG (conjugate gradient), DC (data cube), EP (embarrassingly parallel), FT (Fourier transform), IS (integer sort), LU (lower-upper Gauss-Seidel solver), MG (multi-grid method), SP (scalar pentadiagonal solver), and UA (unstructured adaptive mesh).

The *PARSEC* dataset captures the PARSEC 3.0 benchmarks [14] in their P-Threads implementation. The largest problem class (native) was used in each case. Page table snapshots were taken at 4-14 points during execution at a 10 seconds interval. The specific benchmarks used were: blackscholes, bodytrack, canneal, facesim, ferret, fluidanimate, streamcluster, swaptions, and vips.

The *Legion* dataset captures a version of the HPCG benchmark [23], [34] ported for the Legion run-time system [9], [57]. A central concept in Legion is the *logical region*, an abstraction of multi-dimensional data that decouples the logical structure of the data from its physical layout. This allows the runtime to manage access and layout even with heterogeneous hardware. In this environment, allocations and page-mappings might therefore be substantially different from those seen with run-times without this focus. As we described in the introduction, HPCG on Legion is one instance where substantial speedup is possible in our Nautilus kernel because the identity-mapped address translation that is used results in very few TLB misses. We take page table snapshots during the run of HPCG at one second intervals.

Finally, the *Synthetic* dataset contains page tables constructed that represent an identity mapping ($PPN = VPN$), offset mapping (($PPN = VPN + k$), and segment mapping (a sequence of offset mappings). Identity and offset mappings are two modes of operation of the Nautilus kernel, while segment mapping is what would be achieved if the mmap regions (virtually contiguous regions) in a Unix-style kernel were mapped to physically contiguous regions.

IV. Perfect hashing

The perfect hashing approach to functional address translation involves building a function that makes the search of an inverted page table fast in all cases. Recall that in the traditional page table model, which we will now call the forward page table model, there is a page table entry for each VPN, and that entry contains the PPN. In contrast, in the inverted page table model, the page table contains a page table entry for each PPN, and that entry contains the VPN. A key advantage to inverted page tables is that number of page table entries scales with the amount of physical memory. This is quite unlike traditional forward page tables, where the number of page table entries scales with the size of the virtual address space.[4]

[4]Inverted page tables have a long history and we omit a number of details here. In our analysis, we consider a single inverted page table versus a single forward page table. It is important to note that the inverted page table entry typically also contains an address space (process) identifier. This allows a single inverted page table to be shared by all processes on the machine. In contrast, with traditional forward page tables, a separate page table (hierarchy) is needed for each process. This means that as the number of virtual address spaces grows, the space cost grows as well, instead of remaining fixed to the amount of physical memory as with inverted page tables. It also makes page sharing across virtual address spaces much easier.

Dataset	Page Table Count	Description
Murphy	$771,009$	Databases teaching server; all processes at 15 min intervals over 19 days
Hanlon	$489,569$	Computer systems teaching server; all processes at 15 min intervals over 19 days
Mantevo	13	Mid-run snapshots of each of 13 miniapps
NAS	40	4 snapshots during execution of each of 10 benchmarks
PARSEC	70	4-14 snapshots during execution of each of 7 benchmarks
Legion	221	HPCG port to Legion run-time, 1 second intervals over run
Synthetic	3	Constructed identity, offset, and multiple segment page tables

Fig. 3. Summary of page table snapshots used in our evaluations.

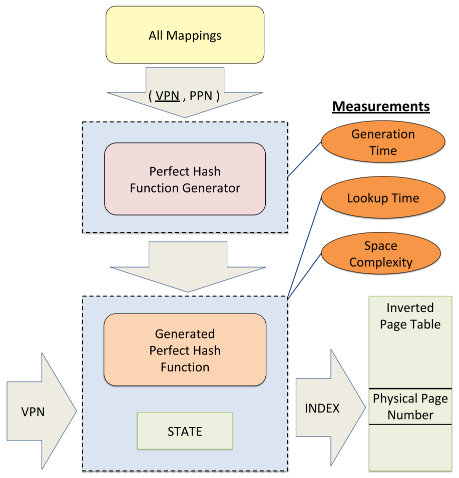


Fig. 4. Functional address translation via perfect hashing.

An important disadvantage of inverted page tables is search. Just as with traditional forward page tables, we must search by VPN. This search is facilitated by a hardware hash function that, given the VPN, finds a good starting point in the inverted page table for the search. In other words, an inverted page table works much like a chaining hash table. When there is no collision, the lookup of the page table entry takes a single step. On the other hand, if there is a collision, a linear search could still ensue, which in the worst case could involve traversing the entire inverted page table.

In the perfect hashing approach, we replace the general purpose hash function with one that is specific to the VPN$\rightarrow$PPN mapping we are encoding in the inverted page table. This bespoke hash function is determined by using a perfect hash function construction algorithm (see below), and then loaded into the hardware. The generated perfect hash function *has no collisions*, and thus the search of the inverted page table is *guaranteed to always involve a single step*. For each lookup, the VPN is hashed, and the result is the index within the inverted page table at which the corresponding entry must be. Figure 4 illustrates this design.

A. Generating perfect hashes

Perfect hashing [28] is a diverse set of algorithmic techniques for producing constant time hash functions with no collisions. These techniques require that the entire set of keys (in our case, VPNs) must be known.[5] Combined with a table, the generated perfect hash function guarantees $\Theta(1)$ lookup in the worst case. One form of perfect hashing, minimum perfect hashing, also guarantees that the space complexity of the generated hash function is minimized, an important consideration for hardware implementations. Theoretically, the space complexity of perfect hashes is linear to the number of keys.

We consider two techniques. The first is the algorithm of Czech, Havas, and Majewski [19], as implemented as the default algorithm (denoted "CHM") of the C Minimum Perfect Hashing (CMPH) library [10], [20]. CMPH is designed to make building minimum perfect hash functions over large keyspaces, for example in a database indexing context, very fast. CHM implements minimum perfect hashing with resulting hash functions having a state size of 8.36 bits per key (or VPN in our case). CMPH's generated functions follow a common template that references tables whose contents are determined by the generation algorithm. A hardware implementation then could combine a fixed hardware component and the ability for it to access the custom tables. Generating minimum perfect hashes (populating the tables) operates in time linear to the number of keys (VPNs). In practice the CMPH implementation generates minimum hash functions for typical processes in milliseconds to 10s of seconds. Using the hash function involves two lookups in the generated tables, which are large and likely must remain in memory. This compares to up to four lookups in forward paging.

The second technique we considered is the algorithm of the GNU Perfect Hash Function (GPERF) library [54]. In contrast to CMPH's algorithms, GPERF's algorithm seeks to build a very fast perfect (although not necessarily minimum) hash functions for smaller keyspaces. Unlike CMPH's algorithms, there is no fixed template for the hash function in GPERF—the generated hash function combines a custom expression and a table. The tables produced by GPERF are much smaller ($\sim$256 entries) than those produced by CMPH. As a consequence, a hardware implementation would require the ability to download the resulting expression in addition to the table. However, the tiny tables could be kept in faster memory (e.g., registers) than those produced by CMPH. Generating perfect hash functions using GPERF is much slower than with CMPH for typical processes.

[5] A variant of this idea, dynamic perfect hashing [21] allows for evolution in the set of keys while maintaining the invariant of no collisions, which we leave for future work.

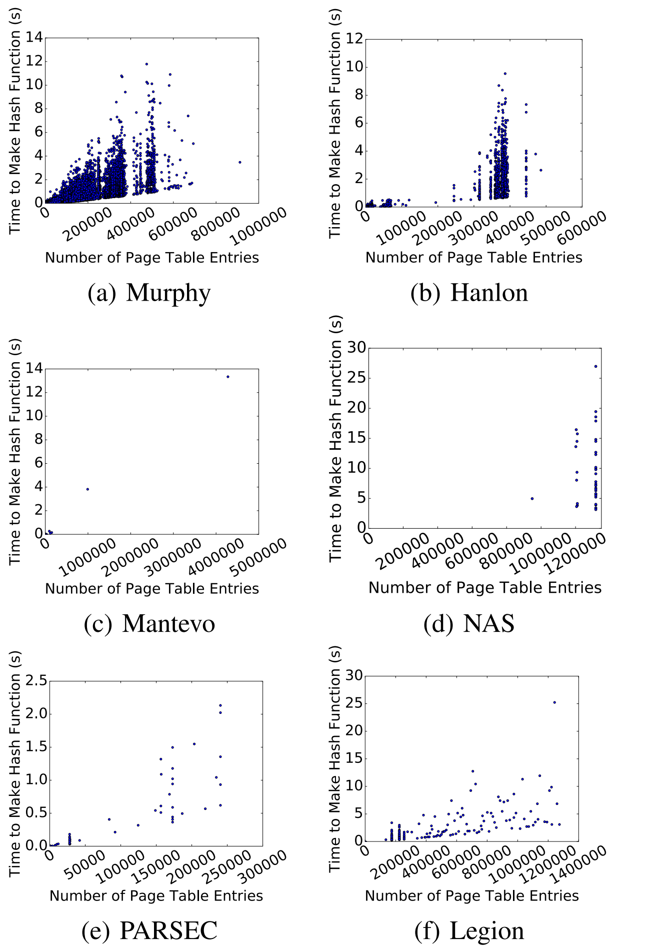


Fig. 5. Minimum perfect hash function generation time using CMPH versus number of page table entries in snapshot.

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.

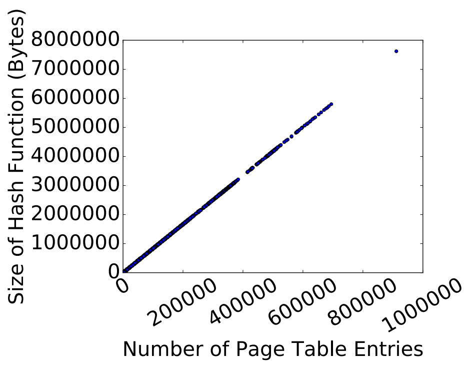


Fig. 6. Minimum perfect hash function space complexity versus the number of page table entries in snapshot. Only the Murphy dataset is shown. All others have the identical straight-line behavior.

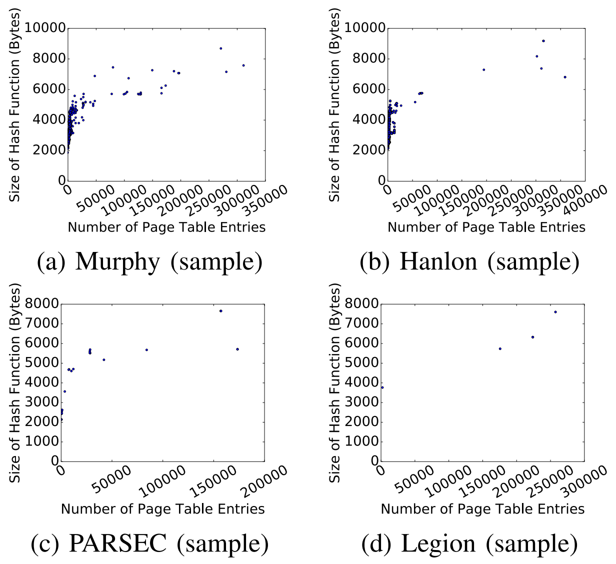


Fig. 7. Non-minimum perfect hash function space complexity versus the number of page table entries in snapshot using GPERF. Mantevo and NAS did not complete.

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 that when it is possible to generate a perfect hash function in GPERF, it is quite compact, and, importantly, the cost tends to be in the code, not the tables.

In considering the lookup costs in a potential hardware implementation, we synthesized a portion of the CMPH output, namely the Jenkins hash [39], [40] that is used internally, for the Intel HARP platform. We hand-coded this hash function in Verilog. The result was compact, requiring $< 1\%$ of available FPGA resources, and could operate in a single cycle of the FPGA core clock. While this is promising, the larger issue with lookup for CMPH/CHM-generated functions is table lookup. In each interesting case, the tables involved are large enough that they would clearly need to be kept in DRAM. Given the operation of the lookup function, we anticipate that this would then require two memory operations during the lookup. At this point, the number of memory operations starts to approach that of the traditional model's pagewalker. This would relegate the CMPH/CHM-based approach to the edge (right hand side of Figure 2.

In contrast, GPERF lookup functions are logic-heavy and memory-light. The tables we produced could readily fit within register-like storage within reconfigurable logic. We did not synthesize the functions, but observations of them suggest that they would likely fit within something like the HARP's FPGA with plenty of room to spare. However, given how incredibly slow GPERF is in generating the function (again, we are using it well outside of its designer's intent), it would be impractical to use a GPERF-based perfect hashing approach expect perhaps in domains where applications run with a stable address space for very long durations, for example in some parts of capability supercomputing.

What seems to be missing to make functional address translation using perfect hashing a practical approach is a perfect hashing function generation algorithm that operates quickly (recall linear time is possible), but produces a hardware rich, as opposed to memory-rich, hash function.

V. Espresso-minimized PLAs

In this approach, we consider the address translation function as a combinational logic function that will be directly instantiated in reconfigurable hardware. This conceptualization is appealing because the resulting logic should perform incredibly quickly (within a single cycle). It should also map to the most basic forms of reconfigurable logic, programmable logic arrays (PLAs), which support only combinational logic, as well as to more advanced forms such as FPGAs. However, it does put the onus on traditional logic synthesis, classic logic minimization, and synthesis techniques. The main issues with this approach are the time necessary to synthesize the address translation function, and the space to represent it.

A. Generating Espresso-minimized PLAs

Figure 8 illustrates our process. The first step is to transform the VPN$\rightarrow$PPN mapping into the inputs and outputs of a logic function. This is simply a truth table representation of a logic expression in which the input columns are the bits of the VPN, and the output columns are the bits of the xPPN. Next, we perform logic minimization on the truth table using the well-known, widely-used Espresso [53] tool. Espresso will apply a logic minimization algorithm to reduce the input and output to a smaller set of terms which maintains coverage of the output (the output remains valid for all inputs provided). Note that the number of rows in the truth table corresponds to the number of active PTEs, not the size of the virtual address space.

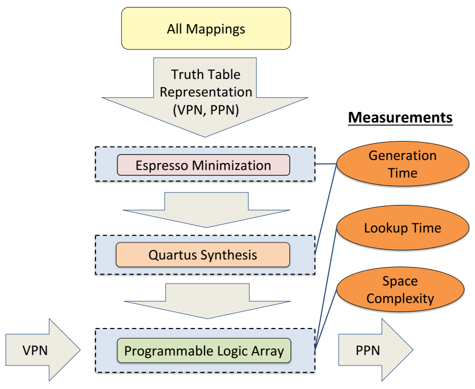


Fig. 8. Functional address translation via Espresso-minimized PLAs.

Given a PLA as the reconfigurable logic target, the output of Espresso can be used to directly configure the logic array as no placement or routing needs to be done. In the case of a target like an FPGA, it is necessary to further transform the Espresso output into Verilog or similar language and then do full synthesis, including placement and routing. The specific tool we use is Quartus 17 which targets the Intel/Altera family of FPGAs. The resulting logic function is entirely self-contained.

B. Study

We applied our process to a sampling of smaller page table snapshots. Our target was an FPGA, specifically an Altera Cyclone IV, which is less than 10% of the size of the HARP platform we mentioned earlier. We measured the process in the following ways:

- Generation time: The combined time to optimize the truth table using Espresso and then synthesize the mapping for the FPGA. The time to actually transform from the snapshot to the truth table input is negligible.
- Space complexity: The size of FPGA block in terms of the FPGA's native logic elements.

We did not measure lookup time because in all cases, the design was synthesized into combinational logic which could complete during the native cycle time of the FPGA (5 ns).

C. Observations

Unsurprisingly, the timescales for Espresso and logic synthesis are such that this approach to translation can only be done well ahead of the runtime of the program. Optimization of the truth table of one process takes on the order of tens of minutes of Espresso time, while the FPGA synthesis to a small FPGA can take closer to an hour. Probably the most

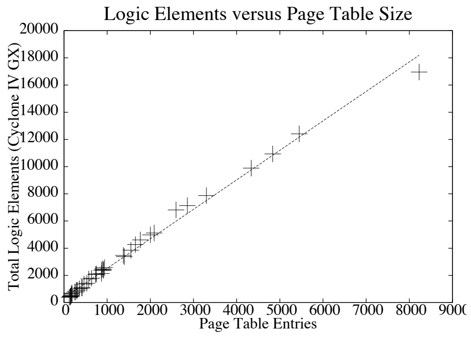


Fig. 9. Size of the Espresso-minimized address translation function on the FPGA as a function of the number of PTEs.

Types:	64 bit word (single type)
Terminals:	0,1,2,9,12, VPN (and components PML4, PDP, PD, PT), and numerous masks
Operators:	not, and, or, xor, aoi, logical shifts and rotates, neg, inc, dec, add, sub, mpy, div, rem, and wordwise and bitwise muxes

Fig. 10. Expression language for creating or evolving pure VPN→PPN functions.

suitable use here would be environments where a process and has a stable page table for a much longer period, perhaps as in some HPC environments such as capability supercomputers.

On a more positive note, Figure 9 shows that the space complexity of the resulting FPGA block is linear in the number of PTEs in a traditional forward page table that the block can replace. The linear relationship is true also of the intermediate Espresso-minimized truth table or PLA. We also considered an alternative model in which there was no Espresso step, but rather the truth table was generated in Verilog and given directly to Quartus. This did not change the results significantly.

While we had initially hoped that a small PLA or FPGA would be sufficient to represent various address translation functions, the linear relationship we actually see suggests that the physical size of the PLA or FPGA needed to support useful mappings is much larger. This rules it out as a TLB replacement, but it could potentially serve as a pagewalker replacement. Another intriguing possibility would be to have the page allocator in the kernel be cognizant of the complexity of the address translation function it is producing, shaping it to be more easily suitable.

VI. Functional Language / Genetic Programming

In this approach, an expression language is designed for the sole purpose of describing pure functions that map from VPNs to PPNs (as well as other components of the PTE). The terminals of the language are a small set of constant values as well as the VPN and its component parts. The operators of the language are designed to be straightforwardly implementable within hardware. The language allows no recursion or iteration of any kind (it is not Turing-complete). As a consequence, any expression in the language (a function for translating VPN to PPN) can be trivially statically analyzed to determine its work and depth complexity, and trivially translated to hardware

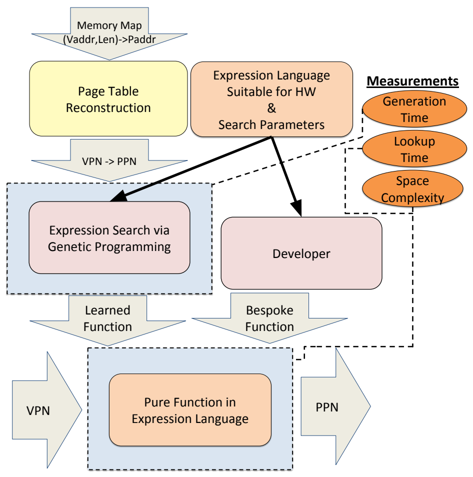

Memory Map
(Vaddr,Len)->Paddr
Page Table Reconstruction
Expression Language Suitable for HW & Search Parameters
Measurements
Generation Time
Lookup Time
Space Complexity
VPN -> PPN
Expression Search via Genetic Programming
Developer
Learned Function
Bespoke Function
VPN
Pure Function in Expression Language
PPN

Fig. 11. Functional address translation by functional language and genetic programming.

to determine area complexity, which is closely tied to work complexity in any case. Figure 10 illustrates the language we designed.

The language could be used directly by the developer or via an indirect, but deterministic construction technique within the OS kernel. For example, simple address space models, such as identity-mapped address spaces (used in Nautilus by default), mapping virtual pages to physical pages at an offset (used to support Multiverse [32] in Nautilus), or classic overlays [55, pp. 222], can be readily expressed with very tiny expressions in the language. We refer to such a function as a *bespoke function*.

Another alternative would be to learn the function for any arbitrary VPN→PPN mapping. Because our expression language involves only a single type, it is straightforward to apply genetic programming [3], [46] to try to find such functions. Genetic programming uses evolutionary pressure toward fitness, combined with mutation and crossover of candidate expressions, to evolve collections of expressions that, over many generations, increasingly become better at approximating the intended mapping. We define fitness as a combination of fit (how close the approximation is) and the size of the expression (work complexity/area). An expression must completely fit the mapping to be selected at the end. Our genetic search process is implemented using the GPC++ framework [27]. The result of this process is a *learned function*.

Figure 11 illustrates the two stage process for both bespoke and learned functions. As with other techniques, we consider the generation time (how long does it take to build an appropriate function?) and the lookup space and time (how expensive is that function?)

A. Study

We developed bespoke functions for the Synthetic dataset. These were straightforward to implement in our language.

We then attempted to apply genetic programming to evolve learned functions for each of the datasets. Unfortunately, none of these were able to complete within a reasonable period (we ran each for a limit of one day before stopping and advancing to the next). In no case, not even for the Synthetic dataset, were we able to learn the function.

B. Observations

Our results are mixed. On the one hand, it is quite clear that our language is sufficient for expressing numerous regular mappings, and that the corresponding functions are tiny and are likely to be fast. Given this, bespoke functions for even the most constrained integration model, Core-TLB, are likely to be practical. At this level, we can think of our results as supporting those of DIY address translation [1]. Pure translation functions, which we designed our expression language for, could push the DIY concept much closer to the processor by allowing the functions to be readily transformed into reconfigurable hardware.

On the other hand, we have had very little success in evolving learned functions, even for mappings that are simple and regular by construction. This is even after many hours of executing the evolutionary process. At this point, we do not believe this is due to a bug in our implementation, limitation in our expression language, or correctness in our fitness function. Instead, we think it is either due to a bad choice of the evolutionary process's parameters (genetic programming is notoriously sensitive to these), or that genetic programming just is not a good fit for this problem. One important distinction of learning an address translation function, compared to other forms of machine learning problems, is that the training data must be captured exactly, while no predictive power is needed at all. That is, we actually need the learning process to "over fit" the data we give it.

VII. Multiplexor Trees

Consider a memory *region* to be the tuple (VPN, PPN, N), which indicates that VPN to VPN+N map to consecutive physical addresses starting at PPN. If run-length encoding an address translation in this manner leads to fewer regions than PTEs, it may serve as a gateway to creating more compact and faster address translation functions. Furthermore, the ability to compactly represent mappings in this manner could be amplified by having the page allocation system use physical contiguity as a page allocation criterion, resulting in fewer (larger) regions. An example of such a system is HPMMAP [45], which already uses contiguity to provide better allocations for NUMA hardware and uses larger pages. This approach reduces the expressiveness required from the synthesized mapping, since many virtual addresses will map to the same physical base page.

Figure 12 shows the potential for this approach. The figure compares the sizes of the three representations across the datasets. Note that the horizontal axis is in log scale. In each figure, the "PTEs" curve is a CDF of the number of active PTEs. The "Original Regions" curve is a CDF of the number of active regions needed to represent the same information in run-length encoded form. Finally, the "Contigified Regions"

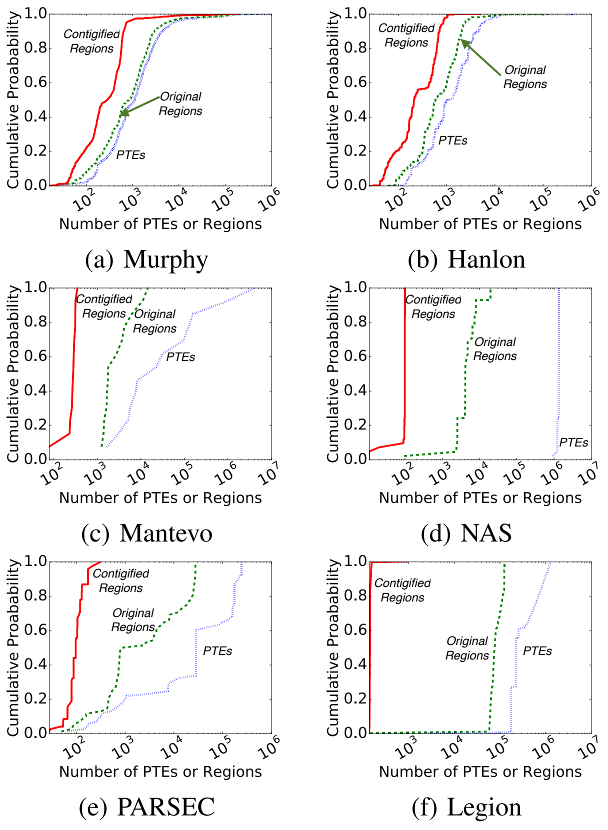


Fig. 12. CDFs of address space sizes as measured by PTEs (rightmost curves) and by virtually (middle) and by both virtually and physically contiguous regions (leftmost). VPN→PPN mappings are more compactly represented by regions that PTEs, and if the page allocator can maintain physical contiguity, the effect is particularly strong.

curve shows the CDF of the number of active regions that would be needed assuming physical contiguity was achieved, that is, that each contiguous chunk of the virtual address space mapped to a contiguous chunk of the physical address space.

Run-length encoding is particularly effective for our HPC benchmarks (Mantevo, NAS, PARSEC, Legion). For example, in Mantevo and NAS, millions of PTEs become thousands of regions; however, even general purpose workloads (Murphy and Hanlon) show 2-5x fewer regions than PTEs across the board. Note also that the "Contigified Regions" curve is shifted dramatically to the left. If the page allocator maintained physical contiguity, 500-1000 regions would be more than sufficient to capture almost all of our workloads (and would capture all the HPC workloads.)

A. Building multiplexor trees

A multiplexor tree is a parallel binary search tree structure in which the VPN is recursively compared against regions by starting address and length. It is similar to the search of the red-black tree representation of the process memory map in the Linux kernel, but the entire tree is in the form of combinational logic suitable for synthesis in a FPGA or similar reconfigurable logic. The leaf nodes are the actual regions. Each interior node of the tree is a multiplexor that outputs to its parent one of the regions produced by its two children depending on whether

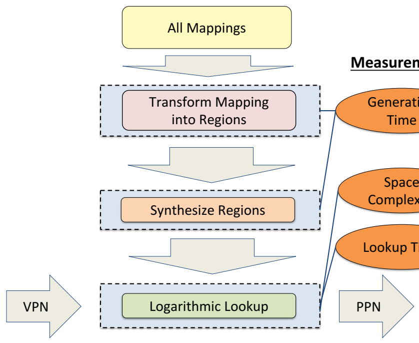


Fig. 13. Address translation by bespoke multiplexor tree.

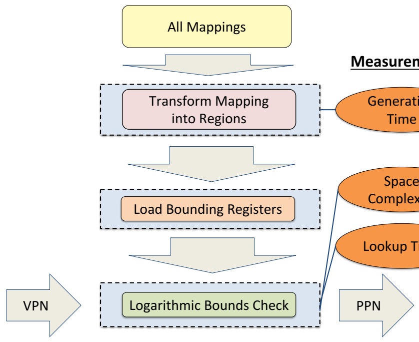


Fig. 14. Address translation by registered multiplexor tree.

the search VPN is $<$ or $\geq$ a region-splitting VPN constructed from the subtree rooted at the node. The VPN is supplied simultaneously to all the interior nodes of the tree. Once the root multiplexor determines the relevant region, the offset into the region is computed with simple addition similar to segment base+offset calculations.

We consider two approaches to producing a multiplexor tree. In a bespoke multiplexor tree, illustrated in Figure 13, all VPN$\rightarrow$PPN 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$\rightarrow$PPN 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.

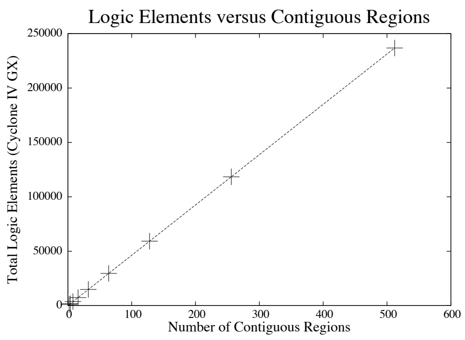


Fig. 15. Space complexity as a function of the number of regions for a bespoke multiplexor tree.

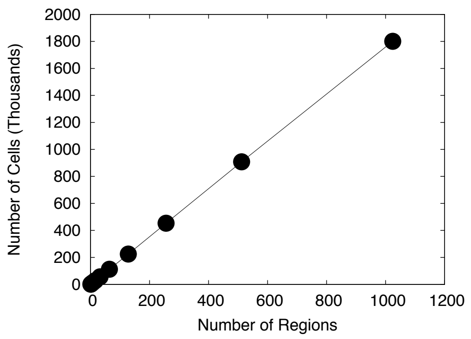


Fig. 16. Space complexity as a function of the number of regions for a registered multiplexor tree.

B. Study

We developed bespoke multiplexor trees for a sample of our page table snapshots, and we produced registered multiplexor trees for a range of sizes. 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.

We did not measure the lookup time of this mapping for the same reasons as given in Section V. We are confident this can be done within a cycle of the FPGA for common cases.

C. Observations

The generation time for both bespoke and registered multiplexor trees is dominated by synthesis within Quartus. For 512 regions, this can take over an hour. Producing the Verilog design itself takes negligible time.

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 $O(logn)$. Given how compactly real address translation functions can be represented as regions (Figure 12), $n \approx 10^3 \ldots 10^4$ can cover a wide range of cases, particularly

	Generation	Space	Lookup
Perfect Hash/IPT (§ IV)			
CMPH	fast	large	slow
GPERF	very slow	small	fast
Espresso/PLA (§ V)	slow	small	very fast
Functional Lang (§ VI)			
Bespoke	depends	small	very fast
Learned (GP)	very slow	small	very fast
Multiplexor Tree (§ VII)			
Bespoke	slow	very small	very fast
Registered	fast	very small	very fast

Fig. 17. Summary of observations and results.

for HPC workloads. With "contigified" regions, $n \approx 10^2$ is sufficient for all the workloads. As a practical matter, at these scales, a multiplexor tree would consume only a small portion of an FPGA or other combination logic, and be able to provide single cycle lookups.

VIII. Prospects

Figure 17 summarizes our study results and observations. At least given the approaches we considered, the prospects for functional address translation are mixed. The prospects and the possible best choices among our models depend on the quadrants of Figure 2.

Because perfect hashing still involves an access to an in-memory inverted page table, it is only suitable for situations that involve a replacement for the pagewalker, Core-Pagewalker and Edge-Pagewalker in our terminology. Even here, it is important to consider the actual costs of the "constant time" generated hash function. Given the comparatively large tables that CMPH-generated hash functions require, these would translate into 1-2 additional memory reads per lookup. At this point, the number of memory references needed is similar to that of a page walk on an ordinary forward page table. The primary motivation to use CMPH-based perfect hashing in a pagewalker would therefore be space—the inverted page table is much more compact than a forward page table. GPERF-based perfect hashing is likely to result in smaller space requirements and faster lookup since its tables for the generated functions are smaller. However, the generation cost of these functions is likely to be prohibitive except when they can be reused.

The future prospects for perfect hashing-based approaches depend very much on the development of algorithms that produce hash functions that combine the generation efficiency of CMPH (CHM) and the representational efficiency of GPERF.

Direct representation of the address translation function via Espresso-minimized PLAs is just one example of an approach where the truth table representation of the function is optimized and embedded in the hardware. Such approaches have the potential of offering the best possible lookup latencies (and small space costs) since the lookup is simply the evaluation of a logic circuit—this lookup could replace a TLB. Our experience with Espresso/PLA bears this out. Possibly, a future processor design could include a PLA block that could be programmed directly from a the logic-minimized design produced by a tool like Espresso.

The Espresso/PLA approach produced circuits whose space requirements, however, were linear in the number of PTEs. These large (but fast) circuits suggest that any PLA block would need to be quite large, which would make it unsuitable for the Core-TLB model, relegating it instead to the Edge-TLB model. Another challenge is the high cost of synthesis. In our experience, it is not the logic minimization cost that is the issue, but mapping to an FPGA. Thus it is possible that having a PLA-block model would reduce generation time to a practical point.

We had high expectations for the functional language approach, especially given results from prior work such as DIY address translation [1]. We did find that if it is possible for the developer to design a bespoke translation function, it is straightforward to encode it in our language, and the synthesized result is likely to be small and fast. However, automatically finding an appropriate function, via genetic programming at least, is simply impractical do within a reasonable period of time. Our genetic programming generation efforts took by far the longest time and CPU cycles of any of our approaches, yet were unable to find any correct functions, even for synthetic functions that a developer could readily write. We do not know what to make of this, or how general the result might be, and so avoid making recommendations.

The multiplexor tree approach performed very well in both of its instantiations. Treating the address space as composed of regions/segments is a powerful abstraction, just as Basu et al [7], [29] found. A register-based configurable translator with enough registers is particularly well suited to the address translations needed by the HPC and parallel applications. Given the number of regions typically needed, this approach could be used even in the Core-TLB model. There is potential to improve on this by synthesizing multiplexor trees that are specific to each given address space, although once again synthesis costs become very problematic.

It is important to note that the multiplexor tree approach is not "the best" we considered. There are serious caveats here as well. In particular, this approach simply cannot encode arbitrary VPN$\rightarrow$PPN mappings as the others are able to.

The high cost of synthesis is a unifying issue for all the approaches we considered. Even a simple Verilog design can take hours to synthesized, without optimizations, on a tool such as Quartus. This pushes designs toward generality—for example, large tables in the perfect hashing approach and region registers in multiplexor tree approach. A synthesis approach that focused on compilation speed would go a along way to making the more specific approaches feasible.

Figure 18 maps the approaches we described into their most suitable quadrants in the space of models (Figure 2). For Core-TLB, the multiplexor tree, particularly the registered variant, is clearly the most sensible, which is no surprise given prior work. Given fast enough synthesis, the bespoke variant would also fit well here. For Core-Pagewalker, both variants of perfect hashing make sense, with the GPERF variant having generation cost as a major caveat.

Recall that by "Edge", we mean a future microarchitecture

Pagewalker	Perfect Hash/IPT (CMPH) (§ IV) Perfect HashIPT (GERF) * (§ IV)	Perfect Hash/IPT (CMPH) (§ IV) Perfect Hash/IPT * (GPERF) (§ IV)
TLB	Multiplexor Tree (Registered) (§ VII) Multiplexor Tree (Bespoke)* (§ VII)	Espresso/PLA * (§ V)
	Core	Edge

Fig. 18. Appropriate FAT mechanisms (with caveats noted by *) for the different quadrants of Figure 2.

in which on-chip caches are virtually indexed and tagged, allowing address translation to happen only when main memory is accessed. This makes it possible for the TLB and Pagewalker to be slower and/or larger. For the Edge-Pagewalker model, the same arguments apply as with Core-Pagewalker, and we believe that this would be the best use of perfect hashing with inverted page tables. GPERF would again be preferable, but only if its long generation times could be either tolerated or overcome. Espresso/PLA would work well for Edge-TLB, given that more space would be available, but this would only make sense if the generation costs could be contained. The functional language approach does not currently fit.

IX. Related work

Address translation has been the subject of numerous studies since the inception of the idea and the introduction of TLBs in computer systems [17]. Over the years, as new workloads surfaced, the research community continues to re-evaluate the overhead of address translation as part of the general memory system characterization. Previous studies have shown that address translation can account for 5-14% of runtime [11], [51], with extreme cases (e.g., in big-data applications and scientific workloads) reaching up to 40-50% of runtime [7], [36], [48], [56]. In virtualized environments the overhead of address translation is further exacerbated, due to the nested address translation from guest to host. Applications in virtualized environments can spend up to 50% of their runtime due to address translation [11], [48], [51].

Numerous techniques have been proposed for reducing address translation overhead, both purely *hardware* [15], [18], [26] and via *hardware/software co-design* [1], [5], [12], [51].

Furthermore, multiple studies [7], [15] have made the observation that pages of larger granularity can significantly increase performance by reducing the address translation overhead. Basu et al. [7] showed that the use of the wrong page granularity can have a significant impact on performance. They observed that Graph500 [47] spends 51% of its runtime on address translation when using 4KB pages, compared to a 10% of runtime when utilizing 2MB pages; however, large page schemes can lead to overheads too, as most implementations limit the number of entries allocated for large pages (e.g., at Intel's Coffee Lake microarchitecture only 4 and 16 entries are allocated for 1GB pages in its first level data TLB and second level shared TLB, respectively).

Though a significant amount of research has studied the patterns [5], [13], [48], [50] and contiguity [51] in address translation, there has been limited investigation on alternatives to the radix tree representation. The majority of research takes a radix tree based translation and TLB caches as a starting point, and proposes improvements upon such designs. Previous works that explore alternatives to radix tree translation are briefly mentioned below. Jacob et al. [38] compared different address translation designs including inverted page tables. Similarly, Yaniv and Tsafrir [58] compared radix tree against hash-based translation, showing that carefully optimized hash-based schemes have higher performance that existing x64 pagewalk-aided designs. Barr et al. [4] compared cached page tables against inverted page tables and direct-mapped Translation Storage Buffers [49]. Finally, Talurri et al. [56] presented an overview of linear, forward-mapped, and hashed page tables and the challenges these designs faced on 64-bit address spaces.

More closely related to our work, Kadayif et al. [42], [44] proposed a hardware/software co-design for direct generation of physical addresses without accessing the TLB, and Basu et al. [7], [29] proposed the use of *direct segments* for mapping a linear portion of the virtual address space, while the remaining address space maintains a page mapping.

Finally, in a complementary work to ours, Hanna et al. [1] present a novel architecture that allows application and virtual machine monitors to specify a custom software-supported address translation mapping.

X. Conclusion

We have defined the concept of functional address translation and then studied it empirically by encoding page tables captured from a wide range of workloads. Four different approaches were considered, including perfect hashing for inverted page tables, Espresso-minimized PLAs that directly represent VPN$\rightarrow$PPN mappings as combinational logic, a functional language suitable for FPGA implementation of address translation functions, and a multiplexor search tree approach that operates over a run-length encoded version of the mappings. The benefits, drawbacks, and recommendations for the approaches are shown in Figures 17 and 18.

While the multiplexor tree approach shows the most promise for current systems, our overall conclusion is that functional address translation has mixed prospects within current systems, but that this could readily change if memory management at the kernel level was restructured, and/or the relationship of address translation and the cache hierarchy was modified. We are currently extending our work in this way.

References

[1] Alam, H., Zhang, T., Erez, M., and Etsion, Y. Do-it-yourself virtual memory translation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2017), ISCA '17, ACM, pp. 457–468.

[2] Bae, C., Lange, J., and Dinda, P. Enhancing virtualized application performance through dynamic adaptive paging mode selection. In *Proceedings of the 8th International Conference on Autonomic Computing (ICAC 2011)* (June 2011).

[3] Banzhaf, W., Nordin, P., Keller, R., and Francone, F. *Genetic Programming: An Introduction: On the Automatic Evolution of Computer Programs and Its Applications*. Morgan Kaufmann, 1997.

[4] Barr, T. W., Cox, A. L., and Rixner, S. Translation caching: Skip, don't walk (the page table). In *Proceedings of the 37th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2010), ISCA '10, ACM, pp. 48–59.

[5] Barr, T. W., Cox, A. L., and Rixner, S. Spectlb: A mechanism for speculative address translation. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)* (June 2011), pp. 307–317.

[6] Barrett, R., Heroux, M., Lin, P., Vaughan, C., and Williams, A. Mini-applications: Vehicles for co-design. In *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing (SC 2011)* (November 2011).

[7] Basu, A., Gandhi, J., Chang, J., Hill, M. D., and Swift, M. M. Efficient virtual memory for big memory servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2013), ISCA '13, ACM, pp. 237–248.

[8] Basu, A., Hill, M. D., and Swift, M. M. Reducing memory reference energy with opportunistic virtual caching. In *Proceedings of the 39th Annual International Symposium on Computer Architecture* (2012), ISCA '12, pp. 297–308.

[9] Bauer, M., Treichler, S., Slaughter, E., and Aiken, A. Legion: Expressing locality and independence with logical regions. In *Proceedings of Supercomputing (SC 2012)* (Nov. 2012).

[10] Belazzougui, D., Botelho, F. C., and Dietzfelbinger, M. Hash, displace, and compress. In *Proceedings of the 17th European Symposium on Algorithms (ESA)* (2009).

[11] Bhargava, R., Serebrin, B., Spadini, F., and Manne, S. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2008), ASPLOS XIII, ACM, pp. 26–35.

[12] Bhattacharjee, A. Large-reach memory management unit caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2013), MICRO-46, ACM, pp. 383–394.

[13] Bhattacharjee, A., and Martonosi, M. Characterizing the tlb behavior of emerging parallel workloads on chip multiprocessors. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques* (Sep. 2009), pp. 29–40.

[14] Bienia, C. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[15] Borg, A., Chen, J. B., and Jouppi, N. P. A simulation based study of tlb performance. In *[1992] Proceedings the 19th Annual International Symposium on Computer Architecture* (May 1992), pp. 114–123.

[16] Choi, Y.-k., Cong, J., Fang, Z., Hao, Y., Reinman, G., and Wei, P. A quantitative analysis on microarchitectures of modern cpu-fpga platforms. In *Proceedings of the 53rd Annual Design Automation Conference (DAC 2016)* (2016).

[17] Couleur, J. F., and Glaser, E. L. Shared-access data processing system, 1968.

[18] Cox, G., and Bhattacharjee, A. Efficient address translation for architectures with multiple page sizes. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2017), ASPLOS '17, ACM, pp. 435–448.

[19] Czech, Z. J., Havas, G., and Majewski, B. S. An optimal algorithm for generating minimal perfect hash functions. *Information Processing Letters 43*, 5 (Oct. 1992), 257–264.

[20] de Castro Reis, D., Belazzougui, D., Botelho, F. C., and Ziviani, N. CMPH - C minimal perfect hashing library. http://cmph.sourceforge.net, 2009.

[21] Dietzfelbinger, M., Karlin, A., Mehlhorn, K., Meyer auf der Heide, F., Rohnert, H., and Tarjan, R. E. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing 23*, 4 (August 1994), 738–761.

[22] Dinda, P., and Guliani, A. Dark shadows: User-level guest/host linux process shadowing. In *Proceedings of the 5th IEEE International Conference on Cloud Engineering* (April 2017).

[23] Dongarra, J., and Heroux, M. A. Toward a new metric for ranking high performance computing systems. Tech. Rep. SAND2013-4744, Sandia National Laboratories, June 2013.

[24] Ekman, M., Stenström, P., and Dahlgren, F. Tlb and snoop energy-reduction using virtual caches in low-power chip-multiprocessors. In *Proceedings of the 2002 International Symposium on Low Power Electronics and Design* (2002), ISLPED '02, pp. 243–246.

[25] Fan, D., Tang, Z., Huang, H., and Gao, G. R. An energy efficient tlb design methodology. In *Proceedings of the 2005 International Symposium on Low Power Electronics and Design* (2005), ISLPED '05, pp. 351–356.

[26] Fan, D., Tang, Z., Huang, H., and Gao, G. R. An energy efficient tlb design methodology. In *Proceedings of the 2005 International Symposium on Low Power Electronics and Design* (New York, NY, USA, 2005), ISLPED '05, ACM, pp. 351–356.

[27] Fraser, A., and Weinbrenner, T. GPC++ - genetic programming C++ class library. http://www0.cs.ucl.ac.uk/staff/W.Langdon/ftp/weinbenner/gp.html, 1997.

[28] Fredman, M. L., Fredman, M. L., Fredman, M. L., Fredman, M. L., Komlos, J., Komlos, J., Komlos, J., Komlos, J., Szemeredi, E., Szemeredi, E., Szemeredi, E., and Szemeredi, E. Storing a sparse table with o(1) worst case access time. In *Proceedings of the 23rd Annual Symposium on the Foundations of Computer Science (FOCS)* (November 1982).

[29] Gandhi, J., Karakostas, V., Ayar, F., Cristal, A., Hill, M. D., McKinley, K. S., Nemirovsky, M., Swift, M. M., and Ünsal, O. S. Range translations for fast virtual memory. *IEEE Micro 36*, 3 (May 2016), 118–126.

[30] Hale, K., and Dinda, P. Enabling hybrid parallel runtimes through kernel and virtualization support. In *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2016)* (April 2016).

[31] Hale, K. C., and Dinda, P. A. A case for transforming parallel runtime systems into operating system kernels (short paper). In *Proceedings of the 24th International ACM Symposium on High Performance Parallel and Distributed Computing, (HPDC 2015)* (June 2015).

[32] Hale, K. C., Hetland, C., and Dinda, P. A. Multiverse: Easy conversion of runtime systems into os kernels via automatic hybridization. In *Proceedings of the 14^{th} IEEE International Conference on Autonomic Computing (ICAC 2017)* (July 2017).

[33] Heroux, M. A., Doerfler, D. W., Crozier, P. S., Willenbring, J. M., Edwards, H. C., Williams, A., Rajan, M., Keiter, E. R., Thornquist, H. K., and Numrich, R. W. Improving performance via mini-applications. Tech. Rep. SAND2009-5574, Sandia National Labs, September 2009.

[34] Heroux, M. A., Dongarra, J., and Luszczek, P. HPCG technical specification. Tech. Rep. SAND2013-8752, Sandia National Laboratories, October 2013.

[35] Hoang, G., Bae, C., Lange, J., Zhang, L., Dinda, P., and Joseph, R. A case for alternative nested paging models for virtualized systems. *Computer Architecture Letters 9*, 1 (January-June 2010).

[36] Huck, J., and Hays, J. Architectural support for translation table management in large address space machines. In *Proceedings of the 20th Annual International Symposium on Computer Architecture* (New York, NY, USA, 1993), ISCA '93, ACM, pp. 39–50.

[37] Intel Corporation. 5-Level Paging and 5-Level EPT. Tech. rep., 2017.

[38] Jacob, B. L., and Mudge, T. N. A look at several memory management units, tlb-refill mechanisms, and page table organizations. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 1998), ASPLOS VIII, ACM, pp. 295–306.

[39] Jenkins, B. Hash functions." algorithm alley". *Dr. Dobb's Journal* (1997).

[40] Jenkins, B. A hash function for hash table lookup (2006). *URL www. burtleburtle. net/bob/hash/doobs. html* (2015).

[41] Jin, H., Frumkin, M., and Yan, J. The Open MP Implementation of NAS Parallel Benchmarks and Its Performance (NAS 3). Tech. Rep. NAS-99-011, NASA, March 1999.

[42] Kadayif, I., Nath, P., Kandemir, M., and Sivasubramaniam, A. Compiler-directed physical address generation for reducing dtlb power. In *IEEE International Symposium on - ISPASS Performance Analysis of Systems and Software, 2004* (March 2004), pp. 161–168.

[43] Kadayif, I., Sivasubramaniam, A., Kandemir, M., Kandiraju, G., and Chen, G. Generating physical addresses directly for saving instruction tlb energy. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture* (2002), MICRO 35, pp. 185–196.

[44] Kadayif, I., Sivasubramaniam, A., Kandemir, M., Kandiraju, G., and Chen, G. Generating physical addresses directly for saving instruction tlb energy. In *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings.* (Nov 2002), pp. 185–196.

[45] Kocoloski, B., and Lange, J. Hpmmap: Lightweight memory management for commodity operating systems. In *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (May 2014).

[46] Koza, J. R. *Genetic Programming: On The Programming Of Computers By Means of Natural Selection*. MIT Press, 1992.

[47] List, T. G. Graph500.

[48] McCurdy, C., Coxa, A. L., and Vetter, J. Investigating the tlb behavior of high-end scientific applications on commodity microprocessors. In *Proceedings of the ISPASS 2008 - IEEE International Symposium on Performance Analysis of Systems and Software* (Washington, DC, USA, 2008), ISPASS '08, IEEE Computer Society, pp. 95–104.

[49] Microelectronics, S. Ultrasparc iii user's manual.

[50] Pham, B., Bhattacharjee, A., Eckert, Y., and Loh, G. H. Increasing tlb reach by exploiting clustering in page translations. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)* (Feb 2014), pp. 558–567.

[51] Pham, B., Vaidyanathan, V., Jaleel, A., and Bhattacharjee, A. Colt: Coalesced large-reach tlbs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2012), MICRO-45, IEEE Computer Society, pp. 258–269.

[52] PK Gupta. Accelerating datacenter workloads, 2016.

[53] Rudell, R. L., and Sangiovanni-Vincentelli, A. Multiple-valued minimization for pla optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 6*, 5 (Sep. 1987), 727–750.

[54] Schmidt, D. C. GPERF: A perfect hash function generator. In *Proceedings of the 2nd Usenix C++ Conference* (April 1990), Usenix.

[55] Silberschatz, A., and Peterson, J. L. *Operating System Concepts (Alternate Edition)*. Addison-Wesley, 1989.

[56] Talluri, M., Hill, M. D., and Khalidi, Y. A. A new page table for 64-bit address spaces. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1995), SOSP '95, ACM, pp. 184–200.

[57] Treichler, S., Bauer, M., and Aiken, A. Language support for dynamic, hierarchical data partitioning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2013)* (Oct. 2013).

[58] Yaniv, I., and Tsafrir, D. Hash, don't cache (the page table). In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science* (New York, NY, USA, 2016), SIGMETRICS '16, ACM, pp. 337–350.