Prediction-Based Superpage-Friendly TLB Designs

Misel-Myrto Papadopoulou* Xin Tong* André Seznec† Andreas Moshovos*

*Dept. of Electrical and Computer Engineering, University of Toronto
Toronto, Canada
Email: myrto@ece.utoronto.ca, trent.tong@gmail.com, moshovos@ece.utoronto.ca

†IRISA/INRIA
Rennes, France
Email: Andre.Seznec@inria.fr

***Abstract*—This work demonstrates that a set of commercial and scale-out applications exhibit significant use of superpages and thus suffer from the fixed and small superpage TLB structures of some modern core designs. Other processors better cope with superpages at the expense of using power-hungry and slow fully-associative TLBs. We consider alternate designs that allow all pages to freely share a single, power-efficient and fast set-associative TLB. We propose a prediction-guided multi-grain TLB design that uses a superpage prediction mechanism to avoid multiple lookups in the common case. In addition, we evaluate the previously proposed skewed TLB [1] which builds on principles similar to those used in skewed associative caches [2]. We enhance the original skewed TLB design by using page size prediction to increase its effective associativity. Our prediction-based multi-grain TLB design delivers more hits and is more power efficient than existing alternatives. The predictor uses a 32-byte prediction table indexed by base register values.**

I. Introduction

Over the last 50 years virtual memory has been an intrinsic facility of computer systems, providing each process the illusion of an equally large and contiguous address space while enforcing isolation and access control. Page tables act as gate-keepers; they maintain the mappings of virtual pages to physical frames (i.e., translations) along with additional information (e.g., access privileges). Except for a reserved part of memory, any code or data structure which currently resides in the computer's physical memory has such a translation.

Page tables are usually organized as multi-level, hierarchical tables with four levels being common for 64-bit systems. Therefore multiple sequential memory references are necessary to retrieve the translation of the smallest supported page size. Hardware *Translation Lookaside Buffers (TLBs)* cache *translations* which are the result of accessing the page table (*page-walk*). A TLB access is in the critical path of each instruction fetch and memory reference; the translation is needed to complete the tag comparison in physically-tagged L1 caches. Thus, a short TLB latency is crucial.

There are technology trends that compound making TLB performance and energy critical in today's systems. Physical memory sizes and application footprints have been increasing without a commensurate increase in TLB size and thus coverage. As a result, while TLBs still reap the benefits of spatial and temporal locality due to their entries' coarse tracking granularity, they now fall short of growing workload footprints. The use of *superpages* (i.e., large contiguous virtual memory regions which map to contiguous physical frames) can extend TLB coverage. Unfortunately, there is a "chicken and egg" problem: some workloads do not use superpages due to the poor hardware support, and no additional support is added as workloads tend not to use them.

The number of page sizes supported in each architecture varies. For example, x86-64 supports only three page sizes: 4KB, 2MB and 1GB, with the last page size (1GB) not enabled in all processors. UltraSparc III supports four page sizes: 8KB, 64KB, 512KB and 4MB, while the MMUs in newer generation SPARC processors (e.g., Sparc T4) support 8KB, 64KB, 4MB, 256MB and 2GB page sizes [3]. Itanium and Power also support multiple page sizes.

While variety in page sizes may cater to each application's memory needs/patterns [4], it may burden the OS/programmer with selecting and managing multiple page sizes. It also makes TLB design more challenging since the page size of an address is *not* known at TLB lookup. This is a problem for set-associative designs as page offset bits cannot be used in the set index. Thus, modern systems support multiple page sizes by implementing multiple TLB structures, one per size, or alternatively resort to a fully-associative TLB structure. Table I lists several modern processor TLB configurations.

TABLE I
Commercial D-TLB designs

Processor	L1 TLB Configuration	L2 TLB Configuration
Intel Haswell [5]	4-way SA split L1 TLBs: 64-entry (4KB), 32-entry (2MB) and 4-entry (1GB)	8-way SA 1024-entry (4KB and 2MB)
AMD 12h family [6]	48-entry FA TLB (all page sizes)	4-way SA 1024-entry TLB (4KB) 2-way SA 128-entry TLB (2MB) 8-way SA 16-entry TLB (1GB)
Sparc T4 [3]	128-entry FA TLB (all page sizes)	
UltraSparc III	2-way SA 512-entry TLB (8KB) 16-entry FA TLB (superpages and locked 8KB)	

Each design has its own trade-offs. Fully-associative (FA) TLBs seamlessly support all page sizes, but are much more power hungry and slower than their set-associative counterparts. Such slow access times are better tolerated in heavily multithreaded systems, such as Sparc T4, where individual instruction latency does not matter as much. Separate per page-size TLBs (e.g., SandyBridge, Haswell) are sized *a priori* according to anticipated page size usage. These structures are all checked in parallel, each using an indexing scheme appropriate for the page size they cache. If a workload does not use some page sizes, the extra lookups waste energy

978-1-4799-8930-0/15/$31.00 ©2015 IEEE

and underutilize the allocated TLB area. Haswell's L2 TLB is the rare example of a commercial set-associative design which supports two page sizes [5]. Unfortunately, the method uses has not been disclosed publicly. Finally, UltraSparc III is representative of designs that just distinguish between 8KB pages and superpages storing the latter in a small FA structure. Workloads that heavily use superpages thrash the small FA TLB.

The goal of this work is to allow translations of different page sizes to co-exist in a single set-associative (SA) TLB, even at the L1 level, while: (1) achieving a miss rate comparable to that of an FA TLB, and (2) maintaining the energy and access time of an SA TLB. The target TLB design should allow elastic allocation of entries to page sizes. That is: (1) A workload using mostly a single page size should be able to use all the available TLB capacity so that it does not waste any resources or be limited by predetermined assumptions on page size usage. (2) A workload that uses multiple page sizes should have its translations transparently compete for TLB entries. An SA TLB will better scale to larger sizes without the onerous access and power penalties of a large FA TLB.

This work first analyzes the TLB behavior of a set of commercial and scale-out workloads that heavily exercise existing TLBs. It finds that some workloads do use superpages heavily. The workloads tend to favor the largest superpage size, while intermediate page sizes rarely appear. Based on these results we propose a lightweight *binary* superpage prediction mechanism that accurately guesses ahead of time if a memory access is to a superpage or not. This enables an elastic TLB_{pred} design that dynamically adapts its super- and regular page capacity to fit the application's needs. We also consider precise page size prediction alternatives [7] for architectures/workloads that may exhibit more variety in their page size usage. We also evaluate the previously proposed but not evaluated Skewed TLB (TLB_{skew}) design [1] which allows translations of different page sizes to coexist. We present TLB_{pskew} that enhances the skewed TLB design with dominant page-size prediction, thus boosting its effective associativity.

Experimental results demonstrate that a 4-way 256-entry TLB_{pred} coupled with a base register value indexed 128-entry bimodal superpage predictor requiring a mere 32 bytes, captures more memory references compared to a slower and less energy efficient 128-way FA TLB. The TLB_{skew} design performs well but suffers from reduced effective associativity in some cases. Fortunately, the prediction-enhanced TLB_{pskew}, with a random-young replacement policy, guided by an 128-entry base register value indexed bimodal predictor performs almost as well as an TLB_{pred}. If implementing true least recently used replacement is possible, TLB_{pskew} outperforms TLB_{pred} albeit only slightly.

The rest of this work is organized as follows: Section II analyzes the TLB behavior of a set of commercial and scale-out applications demonstrating the need for adaptive superpage translation capacity in the TLB. Section III discusses our binary superpage prediction mechanism and Section IV describes how we incorporate it in the proposed TLB_{pred}. Section V presents a summary of the previously proposed TLB_{skew} and the enhanced TLB_{pskew}. Section VI presents our methodology, while Section VII our evaluation results. An overview of related work is presented in Section VIII, followed by our conclusions in Section IX.

II. Analysis of TLB-related Workload Behavior

This section presents a data TLB behavior analysis for a set of commercial and scale-out workloads. Most of the analysis uses full-system emulation of a SPARC system running Solaris (Section VI details the experimental methodology used). However, Section II-C shows, using native runs, that some key observations that motivate the rest of the work also hold true in an x86 system. Section II-A reports statistics characterizing the workload footprints. Section II-B presents how this set of workloads behaves under different TLB designs and also quantifies their area/power trade-offs. We target the data TLB as its performance is much worse than the instruction TLB.

A. Footprint Analysis

Table II shows the average per core number of unique translations for 8KB pages and superpages encountered per workload during the execution of a 16 billion instruction sample on a 16 chip multi-core system. The total memory footprint these correspond to is also shown. In this work a translation/page is considered unique according to the tuple {Virtual Page Number (VPN), Context, Page Size}. Please refer to Section VI for details about the TLB entry format. The system supports page sizes of 8KB, 64KB, 512KB and 4MB.

TABLE II
Footprint Characterization

workloads	Avg. Per-Core 8KB pages	Avg. 8KB Per-Core Footprint (MB)	Avg. Per-Core 512KB pages	Avg. Per-Core 4MB pages	Superpages Per-Core Footprint (MB)
TPC-C1	7071	55	6	544	2179
TPC-C2	17903	140	71	13706	54860
apache	51380	401	50	4	41
canneal	68132	532	0	5	20
ferret	7304	57	0	7	28
x264	2513	20	0	4	16
cassandra	14728	115	0	1469	5876
classification	921	7	0	549	2196
cloud9	28014	219	0	6	24
nutch	7649	60	0	185	740
streaming	53110	415	0	8	32

For the small 8KB pages, Table II shows that the average number of unique pages accessed per-core is one or two orders of magnitude more than existing available TLB capacity. Even though our simulated system supports four page sizes, only the 8KB and 4MB page sizes were prominently used. We observed no use of 64KB page sizes and very few, if any, 512KB pages. The use of superpages varied drastically across the workloads: the OLTP workloads TPC-C1 and TPC-C2 (commercial database systems) and three scale-out applications from the Cloud Benchmark suite, cassandra, classification, and

nutch, use 4MB pages. These workloads can easily thrash an unbalanced TLB design with limited superpage capacity.

B. TLB Miss Analysis

Figure 1 shows the L1 D-TLB Misses Per Million Instructions (MPMI) for TLB designs adapted from Table I. Lower MPMI is better. The series are sorted from left to right in ascending TLB capacity. The first two series model a 48-entry (AMD family 12h-like) and a 128-entry (SPARCT4-like) FA TLB respectively with LRU replacement. The Haswell-like TLB design has been tuned for our system's supported page sizes. It includes four distinct 4-way SA TLBs: a 64-entry TLB for 8KB pages and three 32-entry TLBs for 64KB, 512KB and 4MB page sizes. Finally, the UltraSparc-III-like TLB has a 4-way SA 512-entry TLB for 8KB pages and a 16-entry FA TLB for superpages.

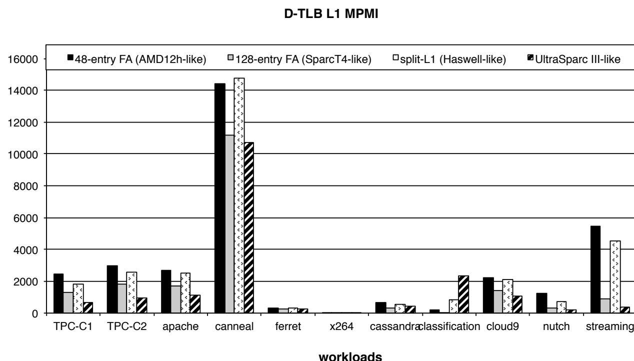

Fig. 1. D-TLB L1 MPMI for different TLB designs

The FA TLBs often have fewer misses than their SA counterparts. Increasing FA TLB size from 48 to 128 entries further reduces MPMI. The UltraSparc-III-like TLB, with its larger capacity for 8KB pages performs best for workloads that mostly use 8KB pages, such as canneal. On the other hand, this design suffers when its small 16-entry FA TLB gets thrashed by the many 4MB pages of a workload like classification whose majority of TLB misses are due to superpages. The split Haswell-based L1 TLBs, with their smaller overall capacity, perform much better for classification but fall short on most others. To summarize the analysis shows that: (1) FA TLBs have a lower miss rate, more so given a larger number of entries. (2) Split-TLB designs are the least preferable choice for these workloads. (3) Capacity can be more important than associativity (e.g., canneal).

Figure 2 plots these TLB designs in a "Dynamic energy per read access" versus "access time" plane using estimates from McPat's Cacti [8]. The preferred TLB design would have the MPMI of the Sparc-T4 (Figure 1), the fast access time of Haswell's split L1 TLBs, and the dynamic read-energy per access of AMD's 48-entry FA TLB. We approach this goal with an elastic set-associative TLB design that uses superpage prediction as its key ingredient.

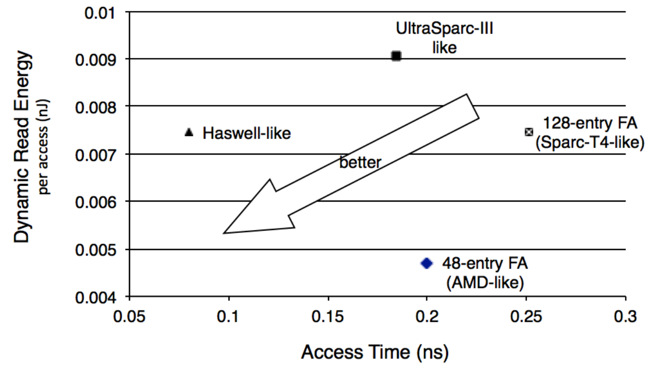

Fig. 2. Access time and Dynamic Energy Trade-Offs

C. Native Runs on an x86

To further demonstrate that superpages are frequent, thus requiring enhanced TLB support, we measured the portion of TLB misses due to superpages during native execution on an x86 system. Table III shows, using performance counters, that superpages can be responsible for a significant portion of TLB misses. Only 4KB and 2MB pages were detected in this system. We ran the workloads for 120 seconds, measuring every 2M TLB misses with *oprofile*. Only a subset of the workloads were available due to software package conflicts.

TABLE III
FRACTION OF TLB MISSES DUE TO 2MB SUPERPAGES (X86)

System Parameters:	Procesor: Intel(R) Core(TM) i7 CPU 860 @ 2.80GHz OS: Linux w/ **enabled Transparent Huge Page support** L1 D-TLBs (4-way SA): 32-entry for 2MB pages & 64-entry for 4KB pages L2 TLB: 512-entry, 4-way SA, shared across instructions and data	
workloads	**% L1 D-TLB Misses**	**% L2 TLB Misses**
canneal	16.4	2.6
cassandra	51.8	14.8
classification	54.5	56.2
cloud9	21.4	33.3

III. PAGE SIZE PREDICTION

The page size of a memory access is unknown during TLB lookup-time. This is a challenge for a set-associative TLB caching translations of all page sizes. Without knowing the page size we cannot decide which address bits to use for the TLB index. This section explains how a page-size predictor can be used to overcome this challenge.

For simplicity, let us assume a system with only two page sizes. A binary predictor, similar to those used for branch direction prediction, would be sufficient here. Using an index available at least a cycle before the TLB access (e.g., PC), the predictor would guess the page size and then the TLB would be accessed accordingly. A TLB entry match could occur only if the predicted size is correct. If this first, *primary,* lookup results in a TLB miss, then either the prediction was incorrect or the entry is not in the TLB. In this case, another *secondary* TLB lookup is needed with the alternate page size. If this also results in a miss, then a page walk ensues.

Most architectures support multiple (N) page sizes. Thus, a binary predictor does not suffice to predict the *exact* page size of an access. In such a system, a page size predictor would have to predict among multiple page sizes [**?**]. It could do so by using wider ($log_2(2*N)$ bits) or multiple saturating counters to predict among the N possible page sizes. Besides the additional hardware and energy costs of this predictor, which may be modest, mispredictions and misses would suffer. On a misprediction up to $N-1$ additional lookups may be needed, if the translation is present in the TLB. These serial lookups hurt performance and energy. We do evaluate such designs in Section VII-F. The rest of this section discusses superpage predictors. Section IV presents the complete TLB$_{pred}$ design.

A. Superpage Prediction

To avoid multiple sequential lookups, we take advantage of the observed application behavior and opt for a binary approach distinguishing between 8KB pages and superpages. Our predictor guesses whether the page is a superpage but it does not guess its exact size. We manage all our superpages homogeneously, as Section IV will show. The proposed predictor uses a prediction table (PT) with 2-bit saturating counters. The PT is a direct-mapped, untagged structure, similar to bimodal branch predictor tables. Each entry has four possible states: (i) strongly predicted 8KB page (P_P), (ii) weakly predicted 8KB page (P_SP), (iii) weakly predicted superpage (SP_P), and (iv) strongly predicted superpage (SP_SP). All entries are initialized to the weakly predicted 8KB state.

For prediction to be possible, the index must be available early in the pipeline. The instruction's address (PC) is a natural choice and intuitively should work well as an instruction would probably be accessing the same data structure for sufficiently long periods of time if not for the duration of the application. However, libraries and other utility code may behave differently. Another option is the base register value which is used during the virtual address calculation stage and thus is available some time before the TLB access takes place. Figure 3 presents the two predictors that use the PC or the base register value as the PT index respectively. In all predictors, the prediction occurs only for memory instructions. In the SPARC v9 architecture, memory instructions have the two most significant bits set to one as shown in the same figure. PT entries are updated only after the page size becomes known: on a TLB hit or after the page walk completes in case of a TLB miss. The predictor tables are never probed or updated during demap or remap operations.

We detail the two PT index types next. In all cases we use the least significant $log_2(\#PTentries)$ bits from the selected field as index, discarding any high-order bits.

PC-based: The first predictor uses the low-order PC bits. This information is available early in the pipeline, as soon as we have identified that this is a memory instruction. A concern with PC-based prediction is that the page size of a given page will be "learned" separately for different instructions. For example, a program that processes different fields of a data structure would do so via different instructions. However,

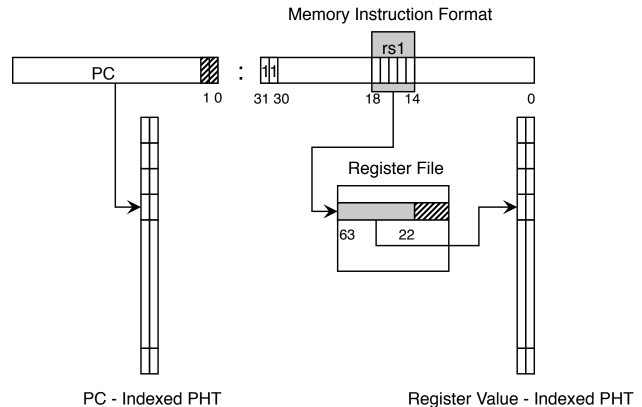

Fig. 3. (a) PC-based and (b) Register-Value based Page Size Predictors

most likely these data fields will all fall within the same type of page (i.e., superpage or 8KB page). Having a PC-based index unnecessarily duplicates this information resulting in slower learning times and more aliasing. Commercial and scale-out workloads often have large instruction footprints, thus putting pressure on PC-based structures

Base Register-Value-Based (BRV-based): Address computation in SPARC ISA uses either two source registers (src1 and src2) or a source register (src1) and a 13-bit immediate. The value of register src1 dominates the result of the virtual address calculation in the immediate case and more than not in the two source register scenario as well (data structure's base address). Therefore we are using the value of source register src1 as an index, after omitting the lower 22 bits to ignore any potential page offset bits (this corresponds to the 4MB superpage size).

To demonstrate how src1 dominates the memory address calculation we provide below a typical compiler-generated assembly of two small loops. The first loop initializes an array and the second sums its elements. The generated assembly of these loops, compiled with g++ with -O3 optimization on a SPARC machine, is as follows:

```
//==========================================
// Loop 1
//==========================================

for (i=0; i< cnt; i++) {
   a[i] = i + 3;
}

/* o2 reg. holds i,
   o3 reg. holds &a[0]
   o4 reg. holds cnt   */

main+0x30:  93 2a a0 02   sll  %o2, 0x2, %o1
main+0x34:  90 02 a0 03   add  %o2, 0x3, %o0
main+0x38:  94 02 a0 01   add  %o2, 0x1, %o2
main+0x3c:  80 a2 80 0c   cmp  %o2, %o4
main+0x40:  06 bf ff fc   bl   -0x10 <main+0x30>
main+0x44:  d0 22 c0 09   st   %o0, [%o3 + %o1]
```

```
//============================================
// Loop 2
//============================================

for (i = 0; i< cnt; i++) {
    sum += a[i];
}

/* o2 reg. holds i.
   o3 and o4 as before. */

main+0x64:   91 2a a0 02   sll   %o2, 0x2, %o0
main+0x68:   d2 02 c0 08   ld    [%o3 + %o0], %o1
main+0x6c:   94 02 a0 01   add   %o2, 0x1, %o2
main+0x70:   80 a2 80 0c   cmp   %o2, %o4
main+0x74:   06 bf ff fc   bl    -0x10 <main+0x64>
main+0x78:   b0 06 00 09   add   %i0, %o1, %i0
```

In both the store (in the branch delay slot) and load instructions, the src1 register (bits 18-14 of the instruction in SPARC-V9) is o3 which is the base address of the array. The src2 register (bits 4-0) is register o1 for the store and register o0 for the load. Thus as expected the value of o3 (array's base address) will dominate.

By using only the base register value, and not the entire virtual address, prediction can proceed in parallel with the address calculation. Accordingly, there should be ample time to access the tiny 32-byte prediction table that Section VII shows is sufficient.

IV. Prediction-Guided Multigrain TLB

The proposed multi-grain TLB, TLB$_{pred}$, is a single set-associative structure that uses two distinct indices: an 8KB-based and a superpage-based index. This "binary" distinction mirrors the observation that there are two prominent page sizes used in the analyzed system (8KB and 4MB). The multi-grain TLB successfully hosts translations of any page size as its tags are big enough for the smallest 8KB supported page size. Figure 4 shows the indexing scheme used for a given TLB size. All superpages, irrespective of their size, share the same indexing bits. Also, Figure 5 shows a potential implementation of the tag comparison for a predicted superpage access. With this indexing scheme all page sizes are free to use all sets. Consecutive 8KB pages and 4MB pages map to consecutive sets. Consecutive 64KB or 512KB pages may map to the same set as they use the same index bits as 4MB pages and thus may suffer from increased pressure on the TLB. As these pages are relatively infrequent, this proves not to be a problem.

While superpage prediction proves highly accurate, correctness must be preserved on mispredictions. Table IV details all possible scenarios. The common case given the high TLB hit rate and prediction accuracy is to have a TLB hit and a correct prediction. A TLB hit during the primary TLB lookup implies a correct page size class (superpage or not) prediction, as each entry's page size information is used for the tag comparison. On a TLB miss, however, there is a degree of uncertainty. A secondary TLB lookup is necessary, this time using the complement page size class. For example, if the prediction was for an 8KB page, the secondary lookup uses the superpage based index. In total, at most two lookups are necessary.

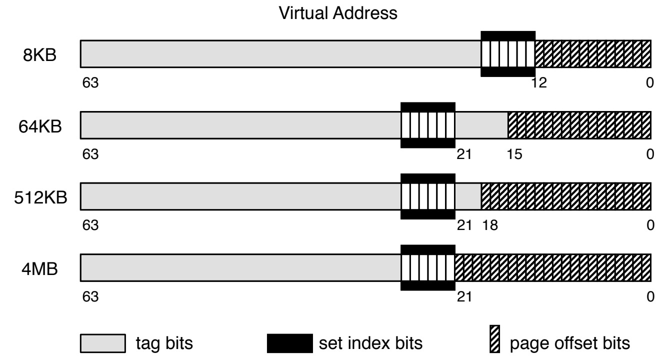

Fig. 4. Multigrain Indexing (512 entries, 8-way associative TLB) with 4 supported page sizes. 6 set-index bits.

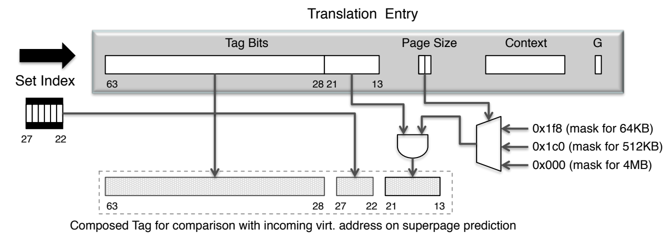

Fig. 5. Multigrain Tag Comparison for Figure's 4 TLB on superpage prediction. Page Size (2 bits) included in every TLB entry.

TABLE IV
PRIMARY TLB$_{pred}$ LOOKUP

TLB Lookup Outcome (w/ predicted page size)	Page size Prediction	Effect
Hit	Correct	Expected common case. No further TLB lookups are required.
Hit	Incorrect	Only possible if both the primary and the secondary lookup probe the same TLB set (i.e., the set index bits for 8KB and 4MB pages are the same) and hardware supports it.
Miss	X (Unknown)	This could either be a misprediction (i.e., we used an incorrect TLB index) or a TLB miss.

Sections IV-A and IV-B discuss how to (a) extend TLB$_{pred}$ for different page size usage scenarios and (b) deal with special TLB operations.

A. Supporting Other Page Size Usage Scenarios

In our analysis we have observed a bimodal page size distribution (i.e., two prominent page sizes) both in SPARC and x86, which motivated our superpage predictor. This was expected for x86-64 which supports 4KB, 2MB and 1GB pages. The 1GB page size, when enabled, is judiciously used so as not to unnecessarily pin such large memory regions. In all cases, the proposed TLB$_{pred}$ correctly works for *any* page size distribution, possibly experiencing increased set pressure for the non dominant page sizes (see Figure 13). We anticipate that the observation that some page sizes dominate will hold in different architectures that also support multiple page sizes.

Precise Page Size Prediction: Assuming that multiple page sizes may be actively used, one solution to avoid conflict misses would be to use a predictor that predicts the exact page size [7]. Thus contiguous pages of all page sizes would map to subsequent sets. The downside is that all mispredicted TLB hits and all TLB misses would pay the penalty of multiple sequential lookups, which could be hefty in systems with a large number of supported page sizes. Section VII-F touches upon such page size usage scenarios further. Table V summarizes the possible outcomes of the secondary TLB lookups for a page size predictor predicting among N possible page sizes. A non-primary TLB lookup that hits in the TLB signals a page-size misprediction. This misprediction overhead is high as it at least doubles the TLB hit latency and may result in having to replay any dependent instructions that were speculatively scheduled assuming a cache hit.

TABLE V
I-TH TLB LOOKUP ($1 < \text{I} \leq \text{N}$); N SUPPORTED PAGE SIZES

i-th TLB Lookup Outcome	Original Page size Prediction	Effect
Hit	Incorrect	i-times TLB lookup latency.
Miss (i < N)	X (Irrelevant)	Repeat lookup with $i+1$ page size.
Miss (i = N)	X (Irrelevant)	True TLB miss. A page walk is in order and thus the increase in latency is, in proportion, small.

Predicting Among Page Size Groups: In SPARC, the two prominent page sizes were the smallest and the largest supported and the difference between the page sizes was not stark. However this might not be the case in other systems. In x86-64 the largest page size is 1GB. Having 2MB pages share the same index as the 1GB pages could result in 512 contiguous 2MB pages competing for the same set, whereas in our system at most 64 contiguous 64KB entries would map to the same set.

A preferable option to precise page prediction, that lowers its worst case penalty, would be to have TLB_{pred} predict among *groups* of page sizes, following the same principle of superpage prediction. These architecture-specific groups should be judiciously selected to minimize potential set pressure due to common indexing. For example, instead of predicting across five page-sizes in Sparc T4, one could predict among the following three groups: (i) 8KB, (ii) 64KB and 4MB, (iii) 256MB and 2GB. Within a group, the index of the largest page-size would be used by the smaller pages. In all cases the TLB entries will have sufficient bits to host the translation of the smallest page size, including each translation's page size information. Finally, for very large page sizes (GB range), that are by default sparsely used and may be limited to mapping special areas of memory (e.g., the memory of graphics co-processors), it might be worthwhile exploring the use of a small bloom filter as a TLB_{pred} addition or use Direct Segments instead [9].

B. Special TLB Operations

MMUs can directly modify specific TLB entries via special instructions. In the Cheetah-MMU that our emulated Ultrasparc-III system uses, it is possible to modify a specific entry in the FA superpage TLB for example to modify locked entries or to implement demap operations. In TLB_{pred}, it is possible that the virtual address of the original TLB entry and the virtual address of the modified TLB entry map to different sets requiring some additional steps. In general, any TLB coherence operation can be handled similar to a regular TLB operation (potentially requiring multiple lookups).

V. SKEWED TLB

A design that supports multiple page sizes in a single structure is the *Skewed TLB*, TLB_{skew} [1]. Unfortunately, no experimental evaluation of its performance exists to date. This section reviews the TLB_{skew} design and explains how we applied it in our evaluated system. In TLB_{skew}, similarly to the skewed associative caches [2], the blocks of a set no longer share the same index. Instead, each way has its own index function. However unlike skewed-associative caches where all addresses see the same associativity, in TLB_{skew} a page maps only to a subset of ways depending on its actual page size and its address.

In more detail, the hash functions are designed in such a way that a given virtual address can only reside in a **subset** of the TLB's ways resulting in a per page size *effective associativity*. The page size of this address's virtual page determines this subset. At lookup time, when the page size is yet unknown, *log2(associativity)* bits of the virtual address are used via a *page size function* which determines that this address can reside in way-subset *X* as page size *Y*. This expected size *Y* information is incorporated in each way's set index, ensuring that both the page offset bits and the page size selection bits for this way are discarded.

Figure 6 shows the set index selection bits of the virtual address for the four page sizes of our baseline system. The set index and page size selection bits are based on a 512-entry, 8-way TLB_{skew}. The set index has six bits, discarding any page size selection bits. These are the indexing functions presented in the original skewed TLB paper [1]. For this TLB organization, virtual address A with bits 23-21 zero can map (i) to ways 0 and 4 if part of an 8KB page, (ii) to ways 1 and 5 if part of a 64KB page, (iii) to ways 2 and 6 if part of a 512KB page, or (iv) to ways 3 and 7 if part of a 4MB page.

The number of supported page sizes is hard-wired in the hash indexing functions and they all have the same effective associativity, two in our example). When an entry needs to be allocated, and a replacement is in order, the effective associativity ways of that given page size are searched for an eviction candidate. In our prior example, if virtual address A belongs to an 8KB page, then only ways 0 and 4 are searched. Since the potential victims reside in different sets an LRU replacement policy could be quite expensive.

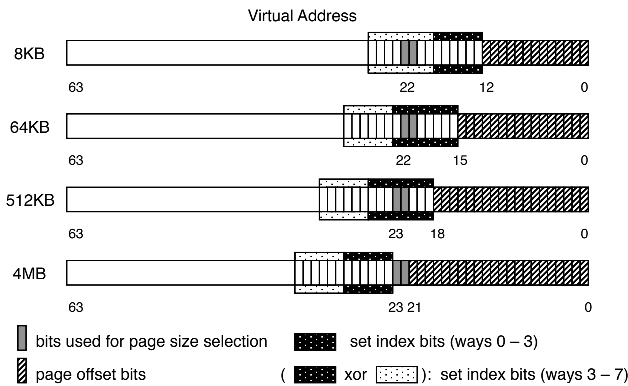

Fig. 6. Skewed Indexing (512 entries, 8-way skewed associative TLB) with 4 supported page sizes

TABLE VI
TLB ENTRY FIELDS

TLB Field (size in bits)	Description
VPN	Virtual Page Number
Context (13)	The equivalent of the Address Space Identifier (ASI) in x86; prevents TLB flushing on a context-switch. The same VPN could map to different page frames based on its context[1].
Global Bit (1)	Global translations are shared across all processes; context field is ignored.
Page Size (2)	Specifies the page size in ascending order: 8KB, 64KB, 512KB and 4MB. Superpages are only allocated in the fully-associative TLB.
PPN	Physical Page (Frame) Number

A. Prediction-Guided Skewed TLB

TLB_{skew} allows a workload to utilize the entire TLB capacity even if it only uses a single page size. However, the effective associativity limits the replacement candidates, causing translation contention.

The default skewed indexing scheme better caters to a uniform use of page sizes, but Section II showed that this is not the common case. There are three considerations: 1) As superpages cover coarser memory regions they are not as many as the 8KB pages. Thus a uniform distribution might not be the best fit. 2) Some workloads, e.g., scientific workloads, mainly use 8KB pages. For them the effective associativity is an unnecessary limitation. 3) For TLBs which use contexts or ASIs (address space identifiers) to avoid TLB shootdowns on a context-switch, the same virtual page could be used by different processes, with all those entries mapping onto the same set. This can apply more pressure due to the imposed effective associativity limit.

We propose enhancing TLB_{skew} with page size prediction with the goal of extending the effective associativity per page size. Specifically, one way of increasing effective associativity would be to perform two lookups in series. In the first we could check for 8KB or 4MB hits and in the second for 64KB or 512KB hits. This way we can use more ways on each lookup per size as there are only two possible sizes each time. The downside of this approach is that it would prolong TLB latency for TLB misses and for 64KB/512KB pages.

We can avoid serial lookups for TLB hits while still increasing effective associativity by using a prediction mechanism. Specifically, we adapt the superpage prediction mechanism so we do not predict between 8KB pages and superpages, but between *pairs* of page sizes. We group the most used page sizes (i.e., 8KB and 4MB) together and the less used page sizes (i.e., 64KB and 512KB) into a separate group. Our binary base-register valued based page size predictor, with the same structure as before, now predicts between these two pairs of pages.

The TLB_{skew} hash functions are updated accordingly so that now only the 22nd bit (counting from zero) of the address is used for the page sizing function. For example, if this bit is zero and we have predicted the 8KB-4MB pair, then this virtual address can reside in ways 2, 3, 6 or 7 as a 4MB page and in ways 0, 1, 4 and 5 as an 8KB page. Similar to TLB_{pred}, if we do not hit during this primary lookup, we use the inverse prediction of a page size pair and do a secondary TLB lookup. Choosing which page sizes to pair together is crucial; in our case the design choice was obvious, as our workloads' page size usage was strongly biased.

VI. METHODOLOGY

This work uses SimFlex [10], a full-system simulator based on Simics [11]. Simics models the SPARC ISA and boots Solaris 8. All experiments are ran on a 16-core CMP system, for 1 Billion instructions per-core, for a total of 16 Billion executed instructions. To achieve reasonable simulation time, we collected D-TLB access traces for all our workloads during functional simulation. We relied on Simics API calls (e.g., probing TLBs/registers) to extract translation information.

In Simics each core models the Cheetah-MMU, the memory management unit for the UltraSPARC-III processors with the D-TLB sizes shown in Table I. Table VI summarizes the relevant fields of each TLB entry. In our system, the TLBs are software-managed. Therefore, on a TLB miss a software trap handler walks the page tables and refills the TLB. This is contrary to x86 systems where the TLBs are hardware-managed. Software-managed TLBs allow for a more flexible page table organization, but at the cost of flushing the core's pipeline and potentially polluting hardware structures such as caches. In the simulated system, the trap handler checks the Translation Storage Buffer (TSB) before walking the page tables. The TSB is a direct-mapped, virtually-addressable data structure, which is faster to access than the page tables. Most TLB misses hit in the TSB requiring only 10-20 instructions in the TLB handler and a single quad load to access the TSB. All accesses are included in our trace, however, the traces should be representative even of systems with hardware page walkers as the number of references due to the TSB is very small compared to the overall number of references and those needed on page walks.

TABLE VII
WORKLOADS

Workload Class/Suite	Workload Name	Description
Online Transaction Processing (OLTP) - TPC-C	TPC-C1	100 warehouses (10GB), 16 clients, 1.4GB SGA
	TPC-C2	100 warehouses (10GB), 64 clients, 450MB buffer pool
Web Server (SpecWEB-99)	Apache	16K connections, FastCGI, worker-threading
PARSEC [12] (native input-sets)	canneal	simulated annealing
	ferret	Content similarity search server
	x264	H.264 video encoding
Cloud Suite [13]	cassandra	Data Serving
	classification	Data Analytics (MapReduce)
	cloud9	SAT Solver
	nutch	Web Search
	streaming	Media Streaming

Workloads: We use the set of eleven commercial, scale-out and scientific workloads summarized in Table VII. These workloads were selected as they are sensitive to modern TLB configurations.

VII. EVALUATION

This section presents the results of an experimental evaluation of various multi-grain designs. Section VII-A shows how accurate our superpage predictors are. Section VII-B demonstrates that TLB$_{pred}$ reduces TLB misses for the applications that access superpages and that it is robust, not hurting TLB performance for the other applications. Section VII-C contrasts the energy of different TLB designs, including our TLB$_{pred}$. Section VII-D evaluates the TLB$_{skew}$ and TLB$_{pskew}$ Skewed TLB designs, while Section VII-E models the resulting overall system performance. Finally, Section VII-F investigates how TLB$_{pred}$ performs under hypothetical, worst case page usage scenarios.

A. Superpage Prediction Accuracy

To evaluate the effectiveness of the superpage predictor we use its *misprediction rate*, i.e., the number of mispredictions over the total number of TLB accesses. Figure 7 shows how the misprediction rate varies over different PT indexing schemes (x-axis labels) and different PT sizes (series). The misprediction rate is independent of the TLB organization. A lower misprediction rate is better as it will reduce the number of secondary TLB lookups. Three predictor handles are shown: (1) PC, (2) base register value and (3) the 4MB-page granularity of the actual virtual address. The last scheme is impractical since it places the prediction table in the critical path between the address calculation and the TLB access. However, it serves to demonstrate that the base register value scheme comes close to what would be possible even if the actual address was known.

[1]The context that should be used for a given translation is extracted from a set of context MMU registers. The correct register is identified via the current address space identifier (i.e., ASI_PRIMARY, ASI_SECONDARY, or ASI_NUCLEUS). For a given machine, the latter depends on the instruction type (i.e., fetch versus load/store) and the trap-level (SPARC supports nested traps).

All prediction schemes perform well. The PC-based index is the worst due to aliasing and information replication. These phenomena are less pronounced for the scientific workloads (e.g., canneal) that have smaller code size. Using the register-value based index performs consistently better than the PC-index. The base register value based predictor is almost as accurate as the exact address-based predictor ("4MB VPN") which demonstrates that the source register src1 dominates the address calculation outcome as expected.

The different series per index explore how the size of the prediction table influences the misprediction rate. The bigger the table the lower the risk of destructive aliasing. With a miniscule 128-entry PT, which requires a meager 32B of storage, the average misprediction rate across the workloads is 0.4% for the base register-value based PT index. Canneal exhibits the worst misprediction rate of just 1.2%. Unless otherwise noted, the rest of this evaluation uses this 32B superpage predictor.

B. TLB$_{pred}$ Misses Per Million Instructions

Our goal was an elastic set-associative TLB design that would have the low MPMI of Sparc-T4 128-entry FA TLB, the fast access time of Haswell's split L1 TLBs, and the dynamic read-energy per access of AMD's 48-entry FA TLB within reasonable hardware budget. Figure 8 compares the MPMI of different TLB$_{pred}$ configurations to the MPMI of commercial-based TLB designs. We vary the TLB$_{pred}$ associativity to have a power of two sets. The results are normalized over the AMD TLB. Numbers below one correspond to MPMI reduction; the lower the better.

The 128-entry FA TLB, targeted for its low MPMI, is 53% better than the smaller 48-entry FA TLB (baseline). Our 256-entry set-associative TLB with its small 32B binary predictor is the TLB$_{pred}$ configuration which meets that goal, being 57.5% better than the baseline. While this configuration uses twice as many entries as the corresponding SPARC T4-like configuration, it is set-associative and, as it will be shown, it is faster and more energy efficient. Compared to the Haswell-like TLB, even the smallest 128-entry TLB$_{pred}$ is considerably better. The 128-entry FA is better than 256-entry TLB$_{pred}$ only for classification. This workload uses hundreds of contexts resulting in many pages with the same virtual address that conflict in the set-associative TLB$_{pred}$. However, overall TLB$_{pred}$'s MPMI is vastly better than the remaining designs and low even for classification.

C. Energy

Figure 9 presents the total dynamic energy (in mJ) for a set of TLB designs. Using McPat's Cacti [8], we collected the following three measurements for every TLB configuration: (i) read energy per access (nJ), (ii) dynamic associative search energy per access (nJ), added to the read energy (i) in case of fully-associative structures, and (iii) write energy per access (nJ). For TLB organizations with multiple hardware structures

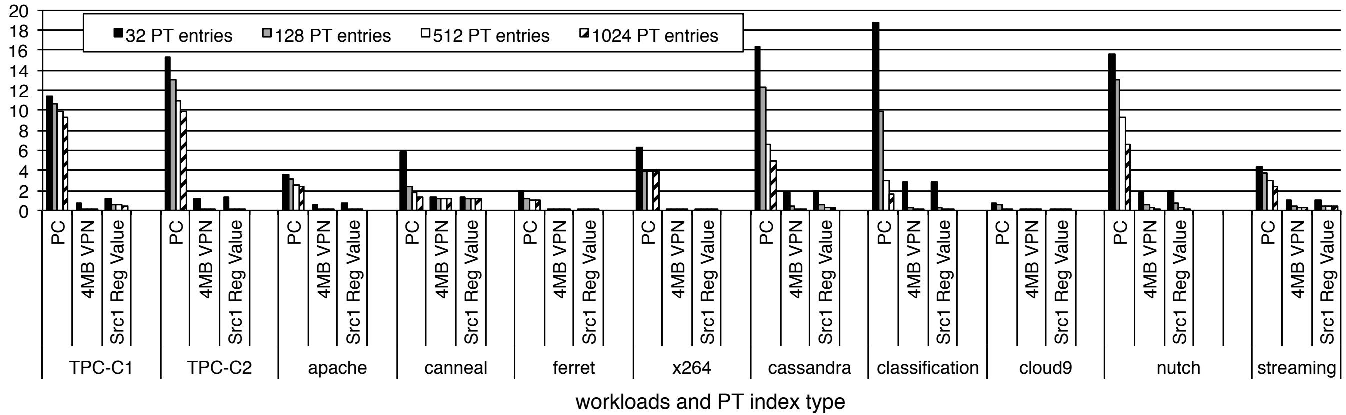

Fig. 7. Superpage Prediction Misprediction Rate (%)

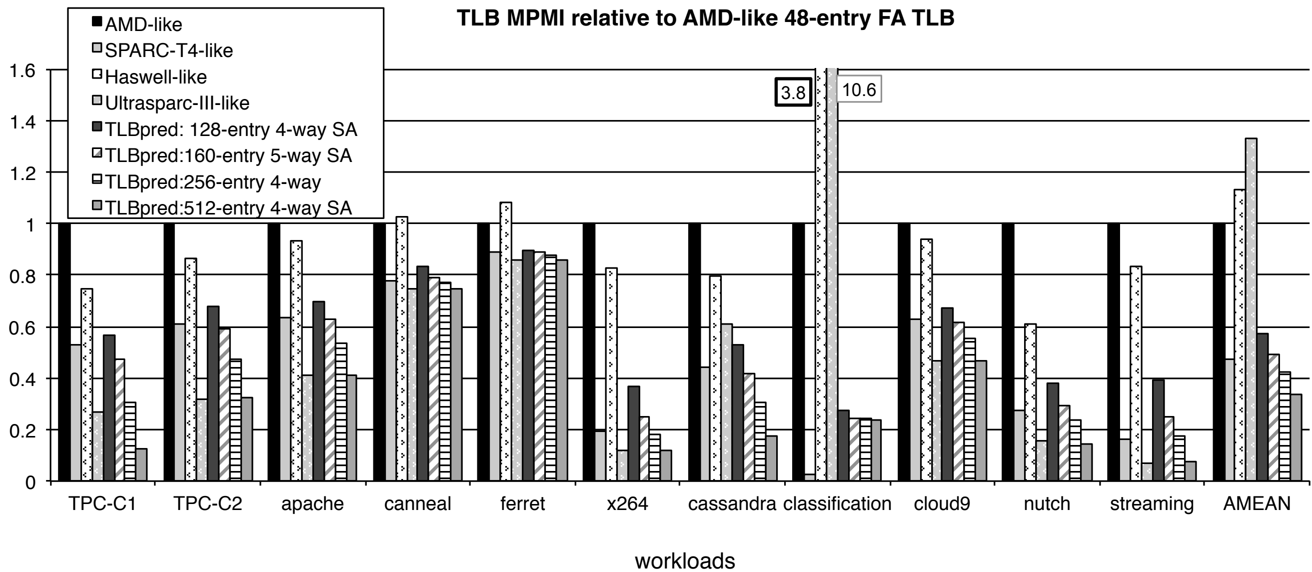

Fig. 8. TLB_{pred} MPMI relative to AMD-like 48-entry FA TLB

(e.g., Haswell) these measurements were per TLB structure. The total dynamic energy of the system was then computed based on the measured TLB accesses (hits/misses) of each structure and workload.

In principle, every TLB access (probe) uses read energy, whereas only TLB misses (allocations) consume write energy. For fully-associative structures (e.g., AMD, Sparc T4), the read energy is the sum of (i) and (ii). For TLB designs with distinct TLBs per page-size (e.g., Haswell), the read energy per probe is the sum of each TLB's read energy as the page size is yet unknown. However, TLB misses only pay the write energy of a single TLB structure, the one corresponding to the missing page's size. The read energy of TLB_{pred}'s secondary TLB lookups was also accounted for, along with the read energy of the 128-entry superpage predictor.

As Figure 9 shows, the UltraSparc-III design has the highest energy. It probes its SA and FA TLBs in parallel and the FA TLB access dominates. The Sparc T4-like FA TLB, with the lowest MPMI of all the designs, also has significantly high energy mostly due to its costly fully-associative lookup. Comparative dynamic energy pays the Haswell TLB-like design due to multiple useless TLB probes of its distinct per-page structures. Our binary page-size prediction mechanism could be employed to avoid this energy waste, serializing these lookups on mispredictions and misses. Finally, the 256-entry TLB_{pred} TLB is the nearest to the target energy of the 48-entry FA TLB, which however has a significantly higher MPMI. The 256-entry TLB_{pred} is the smallest TLB_{pred} design (with lower energy and latency) that meets our MPMI target. Alternatively, the 512-entry TLB_{pred} can yield lower MPMI but at a somewhat higher energy/latency cost.

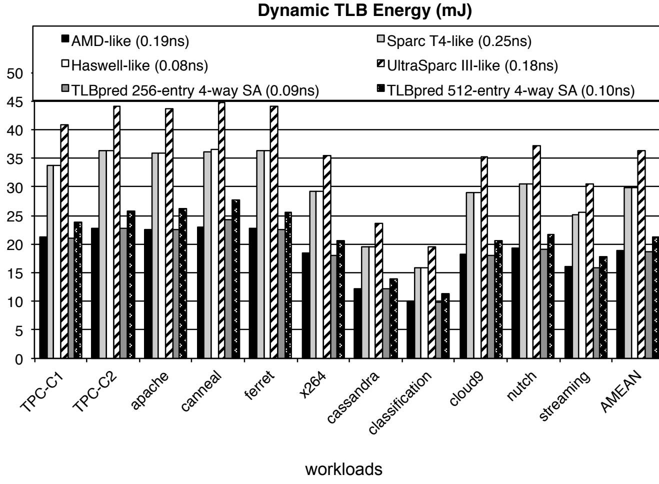

Fig. 9. Dynamic Energy

D. TLB_{skew} and TLB_{pskew} MPMI

This section evaluates different skewed TLB configurations. Figure 10 shows the MPMI achieved by TLB_{skew} and TLB_{pskew} relative to the AMD-like baseline for a 256-entry 8-way skewed-associative TLB. In the interest of space, we limit our attention to 256-entry TLB designs. The first column shows the original TLB_{skew} with a random-young replacement policy. We use the hashing functions described in Section V where the effective associativity for each page size is two [1]. "Random-Young" is a low-overhead replacement policy based on "Not-Recently Used" [14]. A single (young) bit is set when an entry is accessed (on a hit or on a allocation). All young bits are reset when half the translations are young. Upon replacement, the policy randomly chooses among the non-young victim candidates. If no such candidates exist, then it randomly selects among the young entries.

The second column in Figure 10 reports the relative MPMI of the predictor assisted TLB_{pskew} of Section V-A. In any primary or secondary TLB lookup only two page sizes are possible. Therefore the effective associativity for each page size is now four, which proves beneficial in reducing conflict misses. By coupling the 8KB with the 4MB pages during prediction, the predictor achieves a nearly zero misprediction rate (0.07% maximum).

Figure 10 also explores the impact of the replacement policy. The third and fourth columns in the graph depict the TLB_{skew} and TLB_{pskew} design with LRU replacement. Due to the deconstructed notion of a set, LRU would be expensive to implement in hardware [14] compared to the more realistic "random-young". The graph nevertheless reports it as a useful reference point.

Finally, the last column in Figure 10 is our multigrain 256-entry TLB_{pred} design. Our TLB_{pred}, with half the associativity of the skewed designs, reduces MPMI by 46% on average over the AMD-like baseline, whereas TLB_{skew} and TLB_{pskew} with the "random-young" replacement policy reduce it by 36% and 39% respectively. The TLB_{pskew} with the harder to implement LRU policy reduces MPMI by 48% on average.

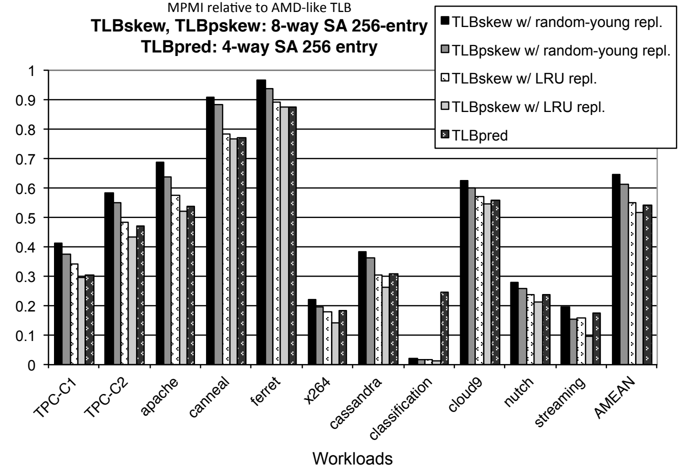

Fig. 10. TLB_{skew}, TLB_{pred}, and TLB_{pskew}: MPMI relative to AMD-like 48-entry FA TLB

E. Performance Model

This section uses an analytical model, as in prior work [15], [16], to gauge the performance impact of the best performing design TLB_{pred}, as the use of software-managed TLBs with the overhead of a software trap handler and the presence of TSB hindered simulation modeling. Saulsbury et al. were the first to use such a model [15] modeling performance speedup as:

$$\frac{CPI_{core} + CPI_{TLBNoOptimization}}{CPI_{core} + CPI_{TLBWithOptimization}} \tag{1}$$

where CPI_{core} is the CPI (cycles per instruction) of all architectural components except the TLB, $CPI_{TLBNoOptimization}$ is the TLB CPI contribution of the baseline and $CPI_{TLBWithOptimization}$ is the TLB CPI contribution under their proposed TLB prefetching mechanism. Bhattacharjee et al. quantify performance impact as "Cycles per Instruction (CPI) Saved" over baseline [16]. This metric is valid irrespective of the application's baseline CPI. Following Equation 1, the $CPI_{TLBNoOptimization}$ can be computed as $MPMI * 10^{-6} * TLB_{MissPenalty}$. Compared to our baseline, the TLB_{pred} has two additional CPI contributors: (1) All page size mispredictions that hit in the TLB pay an extra TLB lookup penalty. (2) All misses also pay an extra TLB lookup penalty to confirm they were not mispredictions. Therefore:

$$\begin{aligned} CPI_{Multigrain} = {} & (MPMI * 10^{-6} * (TLB_{Miss\ Penalty} + TLB_{Lookup\ Time})) \\ & + \frac{Mispredicted\,TLBHits * TLB_{Lookup\ Time}}{Total\ Instructions} \end{aligned} \tag{2}$$

Figure 11 shows the cycles saved by the 256-entry 4-way SA TLB_{pred} over the 128-entry FA SparcT4. We assume a 2-cycle TLB lookup latency for both designs, even though the

FA TLB has a much higher access latency. The x-axis shows different TLB miss penalties. In our system which models a software-managed TLB most TLB misses hit in the TSB as discussed in Section VI. The TLB miss penalties we have observed in our system range from 20 cycles assuming the TSB hits in the local L1 cache, to 60 cycles when the TSB hits in a remote L2 to over 100 cycles when the TSB is not cached. As Figure 11 shows the two designs are comparable in terms of the CPI component due to the TLB. Canneal experiences minimal slowdown, within acceptable error margins, mostly due to its slightly higher misprediction rate.

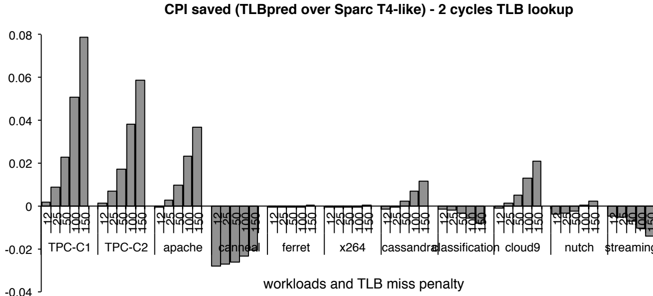

Fig. 11. CPI saved with TLB_{pred}

F. Sensitivity to the Page Size Access Distribution

In our experiments we have seen that there are two prominent page sizes that dominate all TLB accesses. However, there are various factors that can influence the observed page size distribution. For example, (a) the OS's memory allocation algorithm, (b) how fragmented the system is (i.e., there might not be sufficient memory contiguity to allocate large pages), and (c) whether transparent superpage support is enabled or the user requested a specific page size. For completeness, this section explores how TLB_{pred} performs under hypothetical, worse case scenarios.

We chose canneal, the workload with the largest memory footprint, to explore how our proposed TLB design would behave under a different page-size distribution. We used the *ppgsz* utility to set the desired page size for the heap and the stack. Most of the memory footprint is due to the heap. First, we created four canneal spin-offs each with a different preferred heap page size. These configurations have each a different prominent page size as Table VIII shows. Secondly, we created a composite workload with a larger footprint by running two canneal instances, each with a different heap page size (64KB and 4MB). We purposefully selected 64KB and not 8KB to put extra pressure on our TLB_{pred} where consecutive 64KB pages map to the same set potentially resulting in more conflict misses. For the last spin-off we dynamically changed the heap page size throughout execution. This resulted in the highest page size diversity. In all cases we set the page size for the stack to 64KB. The stack is small.

Table VIII reports the resulting distribution of page sizes while Figure 12 shows the TLB miss contribution of each

TABLE VIII
CANNEAL FOOTPRINT CHARACTERIZATION

workloads (heap page size)	Avg. Per-Core 8KB pages	Avg. Per-Core 64KB pages	Avg. Per-Core 512KB pages	Avg. Per-Core 4MB pages
8KB heap	68087	1	0	5
64KB heap	901	9272	0	6
512KB heap	658	1	1160	6
4MB heap	837	1	0	151
4MB and 64KB heap	843	9258	0	152
dynamic heap	39682	8962	62	153

page-size for all our canneal spin-offs for the AMD-like baseline. Unlike the original canneal workload whose misses were solely to 8KB pages, we now observe a different miss distribution. Most of the misses are due to the page size selected for the heap via *ppgsz* as that memory dominates the workload's footprint.

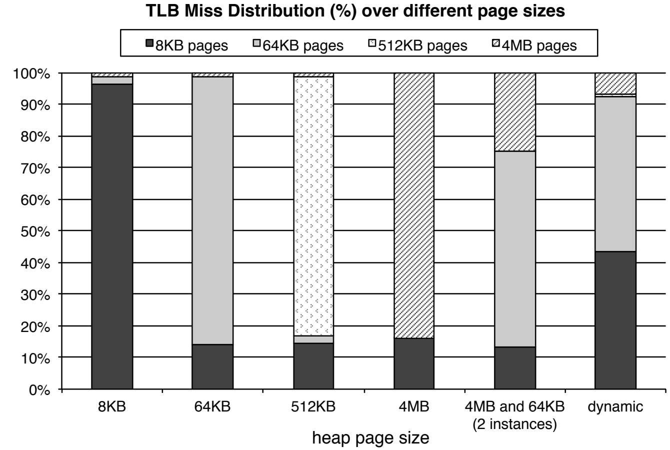

Fig. 12. Canneal Miss Distribution

Figure 13 compares the MPMI achieved via our TLB_{pred} over the AMD-like baseline. The TLB_{pred} is not as good as the SparcT4 TLB for workloads where 64KB and 512KB page sizes dominate but the differences are small. We also modeled a precise page size predictor with larger saturating counters, similar to [7]. Values 0 to 1 correspond to strongly-predicted 8KB page and weakly-predicted 8KB pages, values 2-3 to strongly and weakly predicted 64KB and so on. A correct prediction with an even counter (i.e., strong prediction) results in no updates, while for an odd counter value the state is decremented by one. On mispredictions the counters are incremented by one if the page size is greater than the predicted one or decremented if it is smaller. The last bar in Figure 13 corresponds to this predictor and uses the least significant bits of the predicted VPN for TLB set index. This precise TLB_{pred} design is consistently better then the Sparc T4. As expected, for workloads that use 8KB or 4MB page sizes it performs similar to the superpage prediction based TLB_{pred} in terms of MPMI. For these cases however, which were the observed page size distributions, the precise TLB_{pred} will yield worse latency/energy than our superpage predictor based TLB_{pred} design.

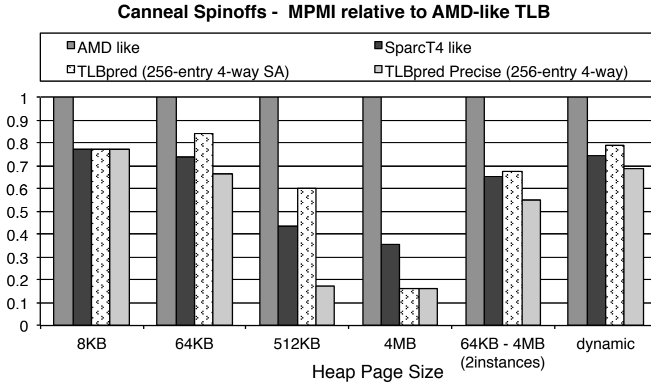

Fig. 13. Canneal MPMI relative to AMD-like TLB

VIII. Related Work

The most closely related to our work is by Bradford et al. which proposes but does not evaluate a precise page size prediction mechanism [7]. The patent lists a variety of potential page size prediction indexing mechanisms, based on PC, register value, and register names and targets exact size prediction. In case of a misprediction this approach would require as many sequential TLB lookups as the number of supported page sizes. Our binary superpage prediction mechanism balances prediction accuracy with misprediction overhead, taking advantage of observed application behavior. Binary prediction and common indexing for different page sizes are the key differences, yielding lower latency/energy. Our TLB_{pred} design seamlessly supports multiple superpage sizes without having to predict their exact size. Moreover, we experimentally evaluate the performance of prediction-guided TLB designs including one that is representative of this design (precise TLB_{pred}).

Other address translation related research either reduces the number of TLB misses or the overhead associated with each TLB miss. A TLB miss can have an onerous performance overhead, up to four times that of a last-level cache miss, due to multi-level page tables. MMU caches can reduce the number of memory accesses required to retrieve a translation by caching part of the page walk [17]. Speculating what the virtual to physical address mapping would be can also hide the page walk latency for operating systems with reservation-based memory allocation [18].

Saulsbury et al. were the first to propose a hardware-based TLB prefetching mechanism based on recency [15]. Kandiraju et al. triggered distance-based prefetches, where a distance is the difference in pages between two consecutive memory accesses [19]. Bhattacharjee et al. studied TLB prefetching in a multiprocessor context and identified two important classes of inter-core TLB misses and proposed prefetching schemes that exploit them [20], [21]. The shared TLB [16] and the Synergistic TLBs [22] were two proposals which facilitated TLB capacity sharing across CMP cores.

Talluri et al. were the first to research the "tradeoffs in supporting two page sizes" [23]. Even though they target 4KB and 32KB pages in a uniprocessor setting, their design observations remain relevant: a fully-associative TLB would be expensive; hosting all translations in a set-associative TLB would require either parallel/serialized accesses with all possible indices or the presence of split TLB structures. The latter is today's design of choice. They also explored the impact of always indexing the set-associative TLB with one of the two supported page numbers showing that indexing with the 32KB page number is slightly worse but generally comparable to "exact" indexing. Our work approximates exact indexing with the use of a binary page size predictor.

An orthogonal approach to superpages is to pack translations for multiple pages within the same TLB entry. Talluri and Hill proposed the "complete-subblock" and the "partial-subblock" TLB designs [24]. Pham et al. proposed CoLT which takes advantage of relatively small scale page contiguity [25]. CoLT's requirement that contiguous virtual pages are mapped to contiguous physical frames is later relaxed [26], allowing the clustering of a broader sample of translations. CoLT coalesces a small number of small pages which cannot be promoted to superpages; it uses a separate fully-associative TLB for superpages. Our TLB_{pred} proposal is orthogonal as it can eliminate the superpage TLB.

Finally, Basu et al. revisited the use of paging for key large data structures of big-memory workloads, introducing Direct Segments i.e., untranslated memory regions [9]. In their workloads' analysis they also observed inefficiency due to limited TLB capacity when large page sizes were used. Our work addresses this inefficiency. TLB_{pred} can (a) complement Direct Segments for those regions that use paging and (b) can do so without OS changes.

IX. Conclusion

In this work we proposed and evaluated two prediction-based superpage-friendly TLB designs. Our analysis of the data TLB behavior of a set of commercial and scale-out workloads demonstrated a significant use of superpages which is at odds with the limited superpage TLB capacity. Thus, we considered elastic TLB designs where translations of all page sizes can coexist without any a priori quota on the capacity they can use. We proposed the TLB_{pred}, a multi-grain set-associative TLB design which uses superpage prediction to determine the TLB set index of a given access. Using only a meager 32B prediction table, TLB_{pred} achieves better coverage and energy efficiency compared to a slower 128-entry FA TLB. In addition, we evaluated the previously proposed Skewed TLB, TLB_{skew}, and augmented it with page size prediction to increase the effective associativity of each page size. TLB_{skew} proved comparable to TLB_{pred}. Finally, we showed that TLB_{pred} remains effective even when multiple page sizes are actively used and also evaluated an exact page size predictor guided TLB.

Acknowledgment

Many thanks to the anonymous reviewers for their comprehensive comments. This research was supported through an

NSERC Discovery Grant, an NSERC Discovery Accelerator Supplement, and an NSERC Strategic Grant. A. Seznec's work was partially supported by the European Research Council Advanced Grant DAL No. 267175.

References

[1] A. Seznec, "Concurrent support of multiple page sizes on a skewed associative tlb," *IEEE Trans. Comput.*, vol. 53, no. 7, pp. 924–927, Jul. 2004. [Online]. Available: http://dx.doi.org/10.1109/TC.2004.21

[2] ——, "A case for two-way skewed-associative caches," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ser. ISCA '93. New York, NY, USA: ACM, 1993, pp. 169–178. [Online]. Available: http://doi.acm.org/10.1145/165123.165152

[3] M. Shah, R. Golla, G. Grohoski, P. Jordan, J. Barreh, J. Brooks, M. Greenberg, G. Levinsky, M. Luttrell, C. Olson, Z. Samoail, M. Smittle, and T. Ziaja, "Sparc t4: A dynamically threaded server-on-a-chip," *IEEE Micro*, vol. 32, no. 2, pp. 8–19, Mar. 2012. [Online]. Available: http://dx.doi.org/10.1109/MM.2012.1

[4] J. Navarro, S. Iyer, P. Druschel, and A. Cox, "Practical, transparent operating system support for superpages," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 89–104, Dec. 2002. [Online]. Available: http://doi.acm.org/10.1145/844128.844138

[5] P. Hammarlund, "4th generation intel core processor, codenamed haswell," August 2013, [Presentation in Hot Chips Symposium, accessed August-2014]. [Online]. Available: http://www.hotchips.org/wp-content/uploads/hc_archives/hc25/HC25.80-Processors2-epub/HC25.27.820-Haswell-Hammarlund-Intel.pdf

[6] AMD, "Software optimization guide for amd family 10h and 12h processors," 2011, [Online; accessed August-2014]. [Online]. Available: http://support.amd.com/TechDocs/40546.pdf

[7] J. Bradford, J. Dale, K. Fernsler, T. Heil, and J. Rose, "Multiple page size address translation incorporating page size prediction," Jun. 15 2010, uS Patent 7,739,477. [Online]. Available: https://www.google.com/patents/US7739477

[8] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009, pp. 469–480.

[9] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 237–248. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485943

[10] N. Hardavellas, S. Somogyi, T. F. Wenisch, R. E. Wunderlich, S. Chen, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk, "Simflex: a fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture," *SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 4, pp. 31–34, Mar. 2004.

[11] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, Feb. 2002.

[12] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[13] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '12. New York, NY, USA: ACM, 2012, pp. 37–48. [Online]. Available: http://doi.acm.org/10.1145/2150976.2150982

[14] A. Seznec, "A new case for skewed-associativity," Internal Publication No 1114, IRISA-INRIA, Tech. Rep., 1997.

[15] A. Saulsbury, F. Dahlgren, and P. Stenström, "Recency-based tlb preloading," in *Proceedings of the 27th annual international symposium on Computer architecture*, ser. ISCA '00. New York, NY, USA: ACM, 2000, pp. 117–127.

[16] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared last-level tlbs for chip multiprocessors," in *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, ser. HPCA '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 62–63.

[17] T. W. Barr, A. L. Cox, and S. Rixner, "Translation caching: Skip, don't walk (the page table)," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 48–59. [Online]. Available: http://doi.acm.org/10.1145/1815961.1815970

[18] ——, "Spectlb: A mechanism for speculative address translation," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11. New York, NY, USA: ACM, 2011, pp. 307–318. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000101

[19] G. B. Kandiraju and A. Sivasubramaniam, "Going the distance for tlb prefetching: An application-driven study," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ser. ISCA '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 195–206. [Online]. Available: http://dl.acm.org/citation.cfm?id=545215.545237

[20] A. Bhattacharjee and M. Martonosi, "Characterizing the tlb behavior of emerging parallel workloads on chip multiprocessors," in *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 29–40. [Online]. Available: http://dx.doi.org/10.1109/PACT.2009.26

[21] ——, "Inter-core cooperative tlb for chip multiprocessors," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV. New York, NY, USA: ACM, 2010, pp. 359–370. [Online]. Available: http://doi.acm.org/10.1145/1736020.1736060

[22] S. Srikantaiah and M. Kandemir, "Synergistic tlbs for high performance address translation in chip multiprocessors," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '43. Washington, DC, USA: IEEE Computer Society, 2010, pp. 313–324. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2010.26

[23] M. Talluri, S. Kong, M. D. Hill, and D. A. Patterson, "Tradeoffs in supporting two page sizes," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ser. ISCA '92. New York, NY, USA: ACM, 1992, pp. 415–424. [Online]. Available: http://doi.acm.org/10.1145/139669.140406

[24] M. Talluri and M. D. Hill, "Surpassing the tlb performance of superpages with less operating system support," in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS VI. New York, NY, USA: ACM, 1994, pp. 171–182. [Online]. Available: http://doi.acm.org/10.1145/195473.195531

[25] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "Colt: Coalesced large-reach tlbs," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-45. Washington, DC, USA: IEEE Computer Society, 2012, pp. 258–269. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2012.32

[26] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh, "Increasing tlb reach by exploiting clustering in page translations," in *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture*, ser. HPCA '14, February 2014.