Improving the Address Translation Performance of Widely Shared Pages

Yousef A. Khalidi
Madhusudhan Talluri

SMLI TR-95-38 February 1995

Abstract:

Operating systems allow multiple processes to share physical objects, e.g., shared libraries, System V shared memory. Many UNIX® implementations allow processes to use different virtual addresses known as aliases to map a shared physical page. Each alias traditionally requires separate page table and translation lookaside buffer (TLB) entries that contain identical translation information. In systems with many aliases, this results in significant memory demand for storing page tables and unnecessary TLB misses on context switches.

This paper first describes a *common-mask* scheme that allows translations from many different virtual address spaces to the same physical address to share a *single* translation entry. It extends the process context id with a bit vector that identifies a set of *common regions* that a process shares with other processes. It requires aliases to use the same virtual address, but aliases with different virtual addresses are still supported in the conventional manner. We then study in detail the implementation and performance effects of applying the common-mask scheme to each level of the translation hierarchy: hardware fully-asociative and set-associative TLBs, memory-based set-associative software level-two TLBs, and finally hashed page tables.

TLB performance improves as processes incur fewer TLB misses on context switches by sharing TLB entries for shared pages. On a set of multi-user benchmarks, we show that the common-mask scheme reduces the number of user TLB misses by up to 50% in a 256-entry fully-associative TLB and a 4096-entry level-two TLB. The memory used to store hashed page tables is dramatically reduced by requiring a single page table entry instead of separate page table entries for hundreds of aliases to a physical page. Common-mask hashed page tables use 97% less memory for an Oracle® Financials database server workload.

The scheme we propose is quite general and applicable in domains other than address translation where multiple keys map to the same data, e.g., virtually-tagged caches, associative processors, and relational databases.

Sun Microsystems Laboratories, Inc.
A Sun Microsystems, Inc. Business
M/S 29-01
2550 Garcia Avenue
Mountain View, CA 94043

email address:
yousef.khalidi@eng.sun.com
madhusudhan.talluri@eng.sun.com

Improving the Address Translation Performance of Widely Shared Pages

Yousef A. Khalidi   Madhusudhan Talluri

Sun Microsystems Laboratories
2550 Garcia Avenue
Mountain View, CA 94043

1 Introduction

Computers that support paged virtual memory [24] typically do address translation through a *translation hierarchy* consisting of *translation lookaside buffers* (TLB) [6,9,21] in or close to the CPU, optional level-two hardware or software TLBs, and, finally, *page tables* in main memory to store virtual-to-physical address translations. A TLB is a cache of the recently used virtual-to-physical translations and reduces average translation time. The tags in a TLB are the virtual page numbers (VPN) and process context ids, and the data consists of physical page numbers (PPN), page attributes (e.g., protection, reference, and modified bits). A page table also stores the same translations and is the backing store for a TLB.

One of the many uses of virtual memory is to allow multiple processes to use the same physical memory pages containing shared data and code. Sharing reduces the working set size [23] of a workload, and enables the workload to fit in a smaller amount of physical memory than if each process had a private copy. Two categories of physical page sharing (Figure 1) are common:

- Read-only sharing of text (code) and data pages. Different instantiations of the same program (e.g., **csh**, **cc**) share text pages, and programs share the same dynamically linked libraries [5] (e.g., **libc**, **libX**, and **libnsl**). Use of the copy-on-write [27] optimization is another example where processes share pages until written to.
- Read-write sharing of data[1] (e.g., System V shared memory). Common applications that exhibit large amounts of read-write sharing are database systems, where many server processes share a large pool of memory representing a buffer cache.

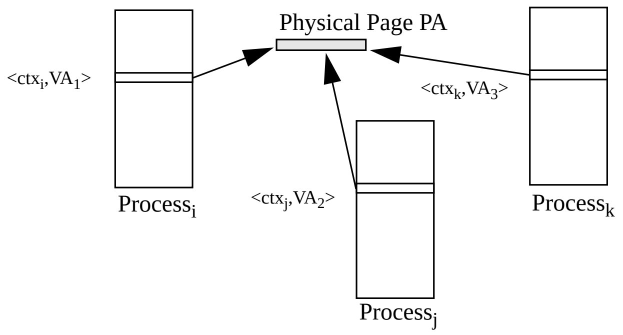

Figure 1. Many mappings to the same physical page

1. Read-write sharing of data is more efficiently supported by hardware and operating systems if the workload is structured as a multi-threaded process instead of using multiple processes.

Many operating systems allow such sharing of physical objects by supporting the establishment of multiple virtual-to-physical address translations for a single physical page. Such virtual addresses are known as *aliases* or *synonyms*. Large commercial and multi-user systems have hundreds of processes that reference the same memory object.

Supporting aliases in a computer system has many problems, especially when there are hundreds of aliases for a physical page. First, current page table designs require storing separate page table entries for each alias, although each entry stores the same translation information. Such multiple copies are inefficient to store, update and keep consistent. Second, TLBs cannot share a single entry for the physical page and incur unnecessary TLB misses on context switches [16]—virtually-tagged caches exhibit similar behavior. Third, virtual addresses used for aliases must be carefully coordinated in machines with virtually-indexed caches [18].

To find out the characteristics of aliases in real systems, we ran the **AimIII** [12] benchmark simulating several hundred users, and a version of the **Oracle® Financials** database on an 8-CPU SPARCServer™ 1000. Some of the results are shown in Table 1. First, only few of the physical pages in the system are read-shared, while most pages are private heap or stack pages. The database system has many read-write shared pages. Secondly, the few shared physical pages are widely shared with hundreds of aliases per physical page. Third, most of the shared mappings have identical page table entries, differing only in the context id. Fourth, processes typically share sets of related pages—pages belonging to the same file or the same group of files.

The *common-mask* scheme proposed in this paper is a way to allow a single translation entry to match the different aliases with multiple tags. The potential benefits of using this scheme in the translation hierarchy are many: First, by using a single TLB entry to map multiple aliases, *TLB reach*—the maximum size of virtual memory mapped by a TLB [15]—is increased, and fewer TLB misses occur. Second, by using a single page table entry to map multiple aliases, page tables occupy a much smaller amount of memory—a significant benefit of which is reduced CPU cache pollution [20]. Third, alias management in the operating system simplifies considerably resulting in a significant reduction in the number of *minor page faults*[2] and TLB shootdowns [2].

Application mix	**Avg. # mappings per physical page**	**Avg. # mappings per shared physical page**	**Largest # of mappings**
Idle System	2.29	27.12	131
Aim3 100 users	2.37	32.36	143
Aim3 500 users	2.58	55.20	564
Oracle Financials 100 users	3.43	105.47	447

Table 1. Mappings per physical page

In Section 3, we develop the general approach of how a single translation entry may be used to store the mapping information for multiple aliases to a physical page. In Section 4, we propose one solution, the *common-mask* scheme, which limits the sharing to a few *common regions*. A common region is a set of physical pages that a process maps using a fixed virtual address and a fixed set of permissions for each physical page in the set. The process context id is extended with a bit vector that identifies the set of common regions that a process shares with other processes. The scheme extends the traditional tag match function by comparing the process context id, $\mathrm{CTX}_{\mathrm{cpu}}$, with the context id stored in the translation entry, $\mathrm{CTX}_{\mathrm{pte}}$, using a simple logical AND function[3] for shared entries, while still requiring the virtual addresses, $\mathrm{VPN}_{\mathrm{pte}}$ and $\mathrm{VPN}_{\mathrm{cpu}}$, be equal. Aliases that do not use the same virtual address are still supported but use different translation entries.

Then we study, in detail, the impact of implementing the common-mask scheme in TLBs and page tables. In Section 5, we show how common-mask fully-associative TLBs may be built. In Section 6, we show how set-associative TLBs could use the common-mask scheme. The context id is used as a parameter to the hash function used to determine the TLB set to lookup, the common-mask scheme would require a more complex index function. In Section 7, we extend hashed page tables [7] with the common-mask scheme and discuss several indirect benefits of eliminating the alias translations from the page tables.

We emphasize that the common-mask scheme can be

2. A minor page fault occurs when the CPU references a virtual address for which a mapping does not exist in the TLB or page tables, but the physical page is in memory. A major page fault occurs when even the physical page is not in memory.
3. Traditional TLBs and page tables use the equality function for matching the context ids.

applied to one or more levels in the translation hierarchy and can coexist with existing mechanisms. A conventional hardware TLB and a common-mask hashed page table, a common-mask fully-associative TLB with a common-mask set-associative level-two TLB and a conventional forward-mapped or linear page table are two of the combinations possible. We also emphasize that the common-mask scheme continues to work, at speed, for private mappings that do not use the common-mask scheme; this is important since only a small portion of the physical pages in a system are widely shared.

In Section 9, we present some preliminary results and study the benefit of using the common-mask scheme in hardware fully-associative and set-associative TLBs, memory-base software TLBs, and hashed page tables. We show that the common-mask page tables uses 97% less memory for a database server running the Oracle Financials. We also show that the common-mask scheme can reduce the number of user TLB misses by up to 50% in a 256-entry fully-associative TLB and a 4096-entry level-two set-associative TLB for a variety of multiprogrammed workloads.

In Section 10, we suggest how the common-mask scheme may be used in other domains—virtually-tagged caches, relational databases, associative processors, and general hash tables.

2 Related Work

The problem of aliases and sharing is not new and there are several ways to address it. First, segments [25] are the elegant, correct way to address the issue. Programs specify a segment and offset within the segment to identify an unique address for the data location, eliminating the need for aliases. A separate mechanism provides protection. However, pure segmented systems have not been popular due to the complicated address generation logic, and 32-bit addresses may not always be sufficient to address all possible segments.

Second, with the coming of 64-bit address spaces, there are several proposals for single address space architectures [3,10]. A single address space architecture disallows aliases, and processes sharing an object use the same virtual address to access the object. While there are many advantages to the single address space approach, there are some problems, including the inability to implement optimizations such as copy-on-write, incompatibility with existing software, and garbage collection of the address space. Our work is not applicable to single address space machines.

Third, in paged-segmented systems, such as HP® PA-Risc™ [30], PowerPC™ [31] and IBM® RT PC [22], the virtual address specifies a (segment-id, offset) tuple and using the segment table translates into an effective global virtual address. This again eliminates aliases but can efficiently support only a limited number of segments with size and alignment restrictions. Operating systems can support aliases in paged-segmented architectures also by having multiple segments for the shared objects, i.e., segment aliases. Operating systems, however, may not allow sharing of portions of objects—e.g., processes read-share only a part of the program data segment while the rest of the segment is private. Using segments properly to eliminate aliases will result in better TLB and page table performance. Our scheme is only marginally effective here and is applicable when sharing segment table entries in global-address space systems with segment aliases. The use of a protection look-aside buffer (PLB) [10] can often reduce the number of segment aliases in a global virtual address space system when processes share the same pages with different protections.

Fourth, some private address space systems support a *global* bit (e.g., [8]). Setting a global bit in either a TLB or page table entry allows all processes to share that entry, e.g., the kernel is often mapped identically in the address space of every process. A global bit can be used only if two very restrictive conditions are satisfied: a) *all* processes in the system must map the physical page at a fixed virtual address with some fixed permissions; and b) no process can use the virtual address to map any other page or use a different set of permissions. Our scheme is more general than the global bit and trivially supports a global bit.

In forward-mapped page tables (e.g., [11]) it is possible for multiple processes to share portions of their page tables by setting the intermediate nodes[4] to point to common nodes in the page table. Yoo and Rogers describe an operating system implementation that shares portions of forward-mapped page tables between processes [20]. Our scheme differs in three ways: First, forward-mapped page table schemes require the sharing to be at a granularity of a node in the page table (e.g., 256K in SuperSPARC™, 4MB in MIPS®), and all of the shared portion of the address space, say 256KB, must be identically mapped in the

4. Some Intel® processors allow even the leaf nodes to specify an indirect pointer and hence use a single PTE for all the aliases.

sharing processes. Our scheme allows sharing of mappings for an arbitrary set of virtual pages. Second, their scheme cannot be applied to TLBs as TLBs are not structured as multi-level tables. Third, their scheme allows processes to share mappings for objects mapped at different (but aligned) virtual addresses in different processes, while our scheme allows sharing only if the same virtual address is used. We have not found this limitation on the virtual address to be a problem because the operating system and user programs usually can choose virtual addresses that work with the common-mask scheme. It is important to note that a common-mask TLB can still be used if forward-mapped page tables are used to share PTEs.

Recent work by Liedtke also addresses the issue of storing mappings for shared pages [32]. Liedtke proposes *virtual aliasing*—shared mappings store a new virtual address to reprobe the page table with—and is attractive when combined with the use of variable-sized mappings. Our scheme does not require storing of additional mappings for shared pages in the page table or affect the TLB miss penalty but an equivalent data structure is required in the operating system to keep track of the common-mask regions. Liedtke also proposes a virtually-indexed cache that supports unaligned aliases, *SF-cache* [33], but does not share a single cache entry as our scheme does (Section 10.1).

TLB performance is also a well-studied area [9,21,28,29] and various schemes proposed to increase TLB reach include use of *superpages* [14] and *subblocking* [15] that many commercial architectures use in their TLBs. These techniques improve TLB performance by increasing the TLB reach of a single TLB entry by storing mappings for multiple virtual pages from a *single* process. These approaches are complementary to the common-mask scheme and can be combined to improve TLB performance further.

3 General Approach

In this section, we describe the general approach to sharing translation entries. In Sections 4 to 7, we show how to build TLBs and page tables using the common-mask scheme.

When the CPU requires a virtual-to-physical translation, it presents the TLB with a context id (obtained from a hardware CTX_{cpu} register) and a virtual address (the TLB uses the virtual page number, VPN_{cpu}, portion assuming a fixed page size). The TLB searches its entries for an entry whose tag (CTX_{pte},

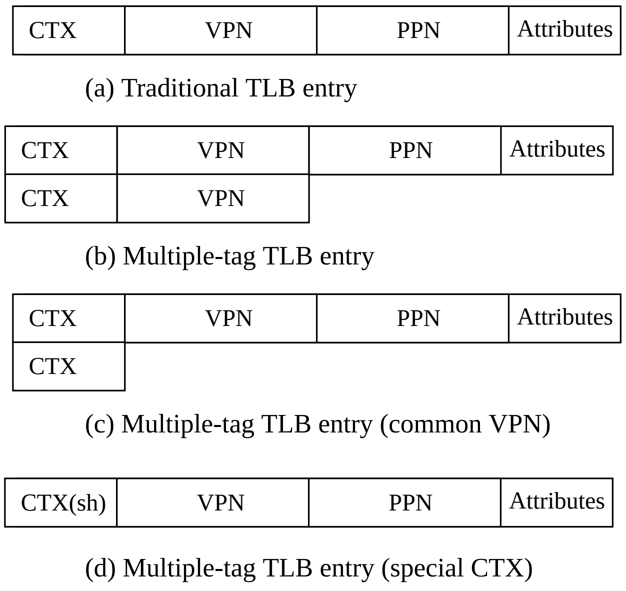
CTX
VPN
PPN
Attributes
(a) Traditional TLB entry
CTX
VPN
PPN
Attributes
CTX
VPN
(b) Multiple-tag TLB entry
CTX
VPN
PPN
Attributes
CTX
(c) Multiple-tag TLB entry (common VPN)
CTX(sh)
VPN
PPN
Attributes
(d) Multiple-tag TLB entry (special CTX)

Figure 2. TLB entry formats

VPN_{pte}), matches the required translation using the function:

$$((CTXcpu = CTXpte) \&\& (VPNcpu = VPNpte)).$$

Figure 2a illustrates a typical TLB entry. For the example in Figure 1, the three processes require three separate TLB entries—the processes have different context ids—and one process cannot use a TLB entry belonging to another process although it maps the same data. The goal of our scheme is to allow a single TLB entry's tag to match with multiple (CTX_{cpu}, VPN_{cpu}) tuples.

The naive approach would be to allow multiple tags for a single data field in each TLB entry—each tag containing a context number and a VPN (Figure 2b). This is the way operating systems often support aliases in operating system data structures, where a list of aliases is attached to the physical page descriptor. However, a hardware implementation of a similar structure is not very attractive for two important reasons. First, for practical reasons, only a small fixed number of tags (aliases) can be supported per TLB entry, whereas in real systems there might be hundreds of aliases for widely shared pages. Second, the hardware cost of implementing tags is significantly more than implementing the data portion. Increasing the number of tags is a significant overhead, and Talluri and Hill [15] show that decreasing the number of tags has significant performance advantages. The naive scheme's matching function, assuming *n* tags per TLB entry, is:

$$\bigcup_{i=1}^{n} ((CTXcpu = CTXptei) \&\& (VPNcpu = VPNptei))$$

One improvement over the naive scheme is to allow only those aliases that use the same VPN to share a single TLB entry. This restriction helps by requiring only a single VPN field in the TLB entry, thus reducing overhead (Figure 2c). Aliases with different VPNs are still allowed, but they have to use separate TLB entries. The operating system or application software must make an effort to choose proper virtual addresses for aliases to get any performance benefits. The TLB entry still requires multiple context number fields for the aliases, an unacceptable overhead and limitation for hardware implementations. Using this improvement, the matching function becomes:

$$\left(\bigcup_{i=1}^{n} (CTXcpu = CTXptei)\right) \&\& (VPNcpu = VPNpte)$$

Our scheme generalizes the matching criterion for the CTX field. Instead of insisting on a strict equality for matching CTX_{cpu} with CTX_{pte}, we allow the use of any arbitrary function, while still requiring that VPN_{cpu} be equal to VPN_{pte} for a match. The match function for each TLB entry in the common-mask scheme becomes:

$$(f(CTXcpu, CTXpte) \&\& (VPNcpu = VPNpte)) \ .$$

There are three interesting features in this simple solution. First, instead of storing multiple context numbers in the tag, the TLB entry stores only a single context number[5] and is much easier to implement (Figure 2d). Second, there is no limitation on the number of aliases that can share a single entry. Third, if the function, *f*, includes the traditional equality check between CTX_{cpu} and CTX_{pte}, it supports the more common non-alias mappings in the same manner as in a conventional TLB.

Therefore, to share a single translation entry for multiple aliases to a physical page, the same virtual addresses must be used for the aliases, with the translations belonging to different processes. The operating system must provide two mechanisms to make this sharing happen. First, the operating system must make a best-effort to allocate aliases to each shared page at the same virtual address in the sharing processes (Section 8). Second, CTX_{cpu} for each process in the system and CTX_{pte} for each shared memory page must be carefully chosen. Mathematically the problem can be stated as follows:

Choose CTX_P per process, CTX_M per memory object page, and a function *f* such that:

$f(CTX_{Pi},CTX_{Mj}) = 1$ for every process P_i that maps the memory object page M_j with the correct protections.[6]

$f(CTX_{Pi},CTX_{Mj}) = 0$ otherwise.

In a real system, the problem of selecting CTX_{cpu} and CTX_{pte} with P processes and C shared pages is non-trivial to solve. There can be thousands of processes and thousands of shared pages. In addition, since the function *f* would have to be implemented in hardware TLB match logic, the function has to be fixed and determined independent of the process mix. Also note that the width of CTX_{cpu} and CTX_{pte} is limited to a few bits—16 to 64 bits. Finally, a real system is dynamic where processes are created and destroyed, and memory objects are mapped and unmapped at a fast pace. This requires an incremental algorithm that makes the problem harder. Given an O(P * C) bits for CTX_{cpu} and CTX_{pte}, a simple solution is possible but the hardware or software to implement it is impractical—a million bit context number might be needed!

We do not attempt to solve the general problem in this paper. Instead, we observe that in a typical computer system there are only a few sets of widely shared pages that the operating system can recognize. In Section 4, we propose a solution that only supports a limited number of common regions, but results in a simple implementation of *f*, and a simple algorithm for assigning CTX_{cpu} and CTX_{pte}.

While this paper concentrates on only the application of this idea for address translation of widely shared pages, the same scheme can be applied to many other domains where multiple keys map to the same data. The common-mask scheme can be applied to the data structure or cache used to store the data for that domain. Examples include virtually-tagged caches, associative processors and relational databases. Some domains may be simple or less dynamic than the address translation scenario and a different algorithm for selection of *f*, CTX_{cpu} and CTX_{pte} maybe more appropriate. Other domains may not require the tag to be divided into two fields and instead use a generic matching function: $(f(KEYcpu, KEYpte))$ where KEY_{cpu} is the lookup key and KEY_{pte} is the key associated

5. The context number stored in the shared PTE need not be equal to any single process' context number.

6. Aliases can share a TLB entry only if they have the same attributes as there is only one data field. Aliases that map the page read-only can share one TLB entry, whereas all the aliases that map the page read-write can share a different TLB entry.

with the data. We do not explore these options further in this paper.

4 Common-mask Scheme

In this section, we describe one possible implementation of the function, *f*, which results in a simple hardware implementation and a simple algorithm for choosing CTX_{cpu} and CTX_{pte}, but is applicable only for a few shared pages. We then extend the scheme to common regions, or collections of pages, which addresses the limitation.

4.1 Description of common-mask scheme

First, the scheme relies on the operating system identifying *n* special (collections of) pages for which PTE sharing is to be used, called *common regions*—the pages that are the most widely shared.

Second, CTX_{cpu}, the per-process id, is extended to include a bit-vector, the *common-mask*, of *n* bits—each bit identifying one of *n* common regions (Figure 3). Thus, the context number of the process specifies the original *m*-bit context id to be used for non-shared mappings and, in addition, identifies the shared regions it maps. The operating system can initialize and update the common-mask as the process maps and unmaps objects.

Third, CTX_{pte} for translations to shared pages stores a bit vector instead of a single process' context number. The bit vector specifies which of the *n* shared regions this page belongs to.[7] A flag identifies whether the translation is to a shared region or is a conventional translation entry.

Fourth, a non-zero result from a simple logical AND of the bit vectors in CTX_{cpu} and CTX_{pte} signals a match. In other words, if the process maps the common region *i* —bit vector in CTX_{cpu} has bit *i* set—and the PTE maps a page that belongs to that region—bit vector in CTX_{pte} has bit *i* set— the logical AND of the bit vectors will have a non-zero value—bit *i* in the result will be set. Otherwise, the result will be zero.

4.2 Structure of CTX_{cpu} and CTX_{pte}

Figure 3 shows the structure of CTX_{cpu} and CTX_{pte}. CTX_{cpu}, the *m*-bit per-process context id is extended by a *n*-bit common-mask bit vector. CTX_{cpu} is also the format of the per-CPU register that identifies the

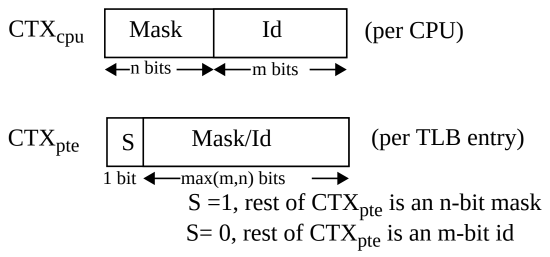

Figure 3. Structure of CTX_{cpu} and CTX_{pte}

currently running process and is updated by the operating system during context-switches. CTX_{pte} replaces the traditional *m*-bits context id in each TLB or page table entry. CTX_{pte} in the common-mask scheme holds either the traditional *m*-bit context id or a *n*-bit common-mask. A shared (S) bit differentiates between the two. For ease of implementation of the TLB, *n* can be set to the same size of *m*, which is typically 10-16 bits wide [8, 11]. By setting $n = m$, this scheme requires only one addition bit per TLB entry over typical TLB designs.

Since there is a single CTX_{cpu} register per CPU, the additional *n* bits are not a significant overhead. As each TLB entry has one CTX_{pte} field, limiting the size of this field results in significant hardware savings.

4.3 The common-mask tag match function

We next define the function $f(CTX_{cpu}, CTX_{pte})$ used during tag matching in a common-mask TLB as:

```
if ($CTX_{pte}$.S) return ($CTX_{cpu}$.mask & $CTX_{pte}$.mask);
else return ($CTX_{pte}$.id == $CTX_{cpu}$.id)
```

Therefore, a TLB entry with the flag S set designates a page in a shared common region and the matching function becomes:
$((CTXcpu \& CTXpte) \&\& (VPNcpu = VPNpte))$.
Otherwise, a TLB entry with the flag S clear maps a page for a single process (CTX_{cpu}.id) and the tag comparison uses the traditional matching criterion:
$((CTXcpu = CTXpte) \&\& (VPNcpu = VPNpte))$.

This function is simple to implement, but has some interesting implementation issues that we discuss in Sections 5 to 7. The function is powerful enough to support arbitrary sharing of multiple common regions by thousands of processes using a single translation entry—all processes that have a bit set in the mask of CTX_{cpu} can use a single translation entry with the same bit set in CTX_{pte}'s mask. In Section 8.1, we

7. A shared mapping can belong to multiple shared regions as explained in Section 4.4.

explain how we choose the masks for CTX_{cpu} and CTX_{pte}. Finally, the function provides an upward compatible path by allowing conventional non-shared translation entries to be treated exactly as before.

4.4 Common regions

A common region specifies a set of physical pages, with each physical page in the set associated with a virtual address and attributes. A process is said to map a common region *iff*: a) the process maps the complete set of pages that belong to the common region; b) each page in the common region is mapped at the correct virtual address associated with the page for this common region; and c) each page is mapped with the correct protections associated with the page for this common region. If a process maps a common region, then the bit vector in CTX_{cpu} has the bit that corresponds to the common region set.

In realistic implementations, the number of allowed common regions is much smaller than the number of shared pages and each common region has a large number of pages. We observe that processes do not normally share individual pages but share objects (which are larger than a page). Therefore, common regions can specify a collection of pages that correspond to an object (or parts of the object) accessed with a given set of permissions, e.g., **/etc/passwd** file with read permission or the text region of **libc** with read / execute permissions.

We further observe that many processes map groups of objects in the same manner. For maximum utilization of the limited number of common regions, we can specify common regions to map a collection of (parts of) objects, e.g., the common region C_0 may designate <**libc** mapped at VA_1 read-only; **libX** mapped at VA_2 read-only; **libnsl** mapped at VA_3 read-only>.

Further, a given physical page can be part of two or more common regions and still share a single translation entry, e.g., C_0 may designate <**libc** mapped at VA_1, **libnsl** mapped at VA_2> and C_1 may designate <**libc** mapped at VA_1, **libX** mapped at VA_3>. The page table entries for pages of **libc** will have two bits set in CTX_{pte}—the bits corresponding to common regions C_0 and C_1. Processes which map either C0 or C1 will match on the same CTX_{pte} for **libc** pages.

By allowing a common region to be defined as a single page, a single object or as a collection of (parts of) objects, the operating system has significant flexibility in either statically or dynamically defining the optimal set of pages for the limited number of common regions.

5 Common-mask Fully-associative TLBs

In fully-associative TLBs, the tags of all the entries are searched in parallel by tag comparators associated with each entry. Adding the common-mask scheme to a fully-associative TLB is relatively simple—change the tag comparator for each entry to implement the common-mask match function. When inserting a new entry into the TLB, any entry in the TLB can be used.

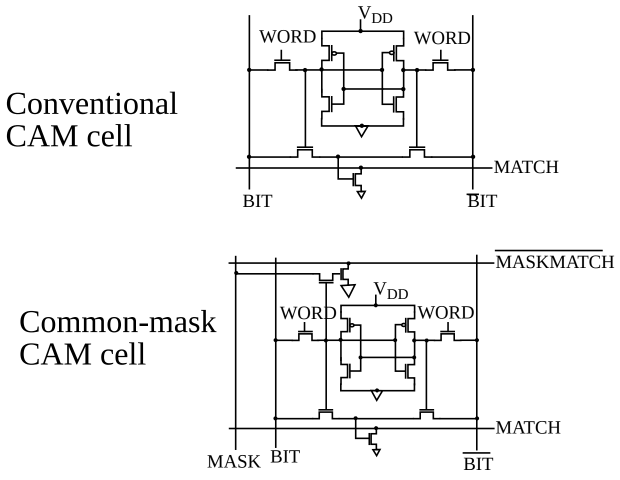

Figure 4. Conventional vs. Common-mask CAM cell

Figure 4 compares a conventional content-addressable memory (CAM) cell with one possible implementation of the CAM cell used in a common-mask TLB. Figure 5 shows how regular CAM cells can be put together to compare an 8-bit PID with the contents of the CAM. $\overline{\text{PIDi}}$ is sent out on the BIT line of cell i. PIDi is sent out on $\overline{\text{BIT}}$ line. If there is any mismatch, the precharged MATCH line is discharged.

The second part of Figure 5 shows how the common-mask CAM cells can be put together to compare CTX_{cpu}—an 8-bit PID and an 8-bit MASK—with the contents of the CAM. The PID portion of CTX_{cpu} is driven on the bit lines as before and the mask portion of CTX_{cpu} is driven on the MASK lines (no need for inverted input). The array of cells outputs two signals: MATCH and $\overline{\text{MASKMATCH}}$—which is inverted to produce MASKMATCH. The shared (S) bit indicates whether the entry stores a mask or a PID, and selects one of the two match signals to produce the context match signal. The context match signal should be combined with the VPN match signal to declare a hit/miss for that TLB entry—not shown here.[8]

There are three key implementation differences between building a conventional and a common-mask TLB. First, conventional CAMs have a single match line that combines the VPN match and CTX match, whereas the common-mask scheme requires separate match lines and circuitry to combine the two. This complicates the design and layout in custom VLSI where the pitch of the CAM cell is restricted. Second, the common-mask TLB requires more transistors per cell and has more lines crossing the cells, vertically and horizontally, which increases the area overhead. Third, the circuitry for combining the different match signals will add to the critical time slightly, but to minimize any impact on TLB access time, this circuitry can be integrated with the drivers that propagate the MATCH signal.

Since it may be expensive to modify all the TLB entries to support both conventional and shared mappings, variations of the above are possible by implementing different types of tag comparators in the TLB—conventional-only, common-mask-only, combined conventional/common-mask (Figure 4). Since a fully associative TLB has an individual comparator for each TLB entry, this can be implemented. Alternatively, separate TLBs can be built with only entries for a single kind of comparator in each TLB—the TLBs can be accessed in parallel, and the output of the different TLBs can be combined with a multiplexor.

The first variation would be a TLB with a mixture of conventional-only and combined common-mask comparators—only entries with the common-mask comparators can store shared entries. Typically, since only a small fraction of the mappings present in the TLB are of the shared variety, this might be a reasonable compromise.

Another variation would be to build a comparator that includes only the logical AND function and does not support private mappings, a common-mask-only comparator—a simpler implementation than either shown in Figure 4. However, this solution will result in unused resources, as some workloads may not use shared mappings and cannot use some TLB entries.

Therefore, fully-associative TLBs can implement the common-mask easily, but it has some area and cycle time costs which must be considered.

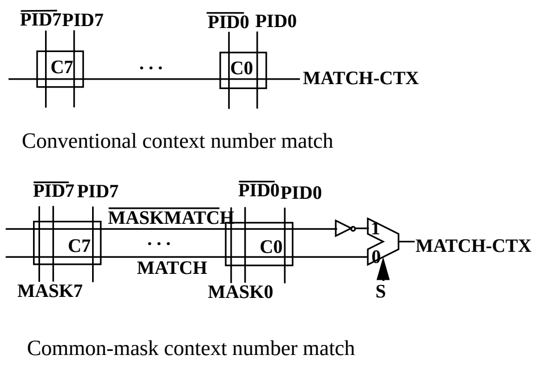
$\overline{\text{PID7}}$ PID7
$\overline{\text{PID0}}$ PID0
C7
C0
MATCH-CTX
Conventional context number match
$\overline{\text{PID7}}$ PID7
$\overline{\text{PID0}}$ PID0
$\overline{\text{MASKMATCH}}$
MATCH
C7
C0
1
0
MATCH-CTX
MASK7
MASK0
S
Common-mask context number match

Figure 5. Conventional vs. Common-mask context matching for fully-associative TLBs

8. Also not shown here is the standard circuitry in a CAM array—precharge circuitry, wordline drivers, sense amplifiers, match signal amplifier—they do not differ from conventional TLB designs.

6 Common-mask Set-associative TLBs

In set-associative TLBs, a subset of the tags are read out from the tag array and compared using the match function—the number of tags read is the associativity. Using the common-mask scheme requires two modifications—the tag comparator logic and the index function used to select the subset of tag.

Figure 6 compares the implementation of the tag compare logic [19] in a set-associative TLB for conventional and common-mask TLBs. The logic is similar to the CAM design, but the impact is much smaller since there are only a few comparators in a TLB and they do not have very tight layout constraints. Again, the logic to combine the three match signals can be integrated with drivers that propagate the MATCH signal.

Modifying the index function of a set-associative TLB to include the common-mask is more complicated. Both CTX_{cpu} and VPN_{cpu} are used, with a hash function, h(CTX,VA), to select the TLB set for lookup. Since shared PTEs match multiple CTX_{cpu}s, there is no unique set that will match all contexts that share a PTE. One solution would be to choose h and CTX_{cpu} such that h(CTX,VA) is same for all processes sharing a common region. As mentioned before, we do not have an incremental algorithm for this. The following solution uses a multiple probe hash function for lookups:

- Insertion of a TLB entry with tag <CTX_{pte}, VA_{pte}> uses:

  If CTX_{pte}.S = 0, then use $h(CTX_{pte}.id, VA_{pte})$

  If CTX_{pte}.S = 1, then use $h(0, VA_{pte})$

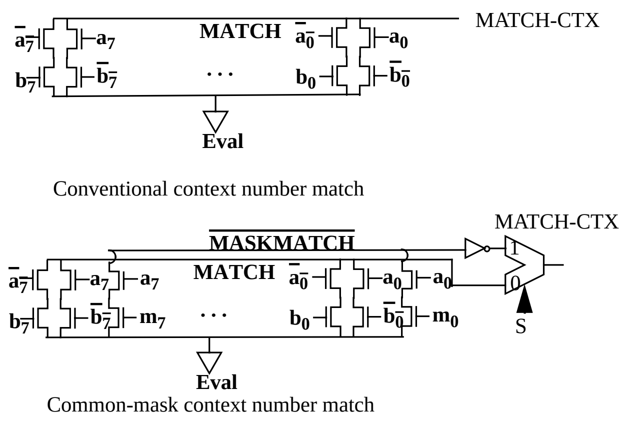
MATCH-CTX
MATCH
Eval
Conventional context number match
MASKMATCH
MATCH-CTX
MATCH
S
Eval
Common-mask context number match

Figure 6. Conventional vs. Common-mask context matching for set-associative TLBs

- Lookup of <CTX_{cpu},VA_{cpu}> uses a two-step process: First index the TLB using $h(CTX_{cpu}.id, VA_{cpu})$. If this fails, index using $h(0, VA_{cpu})$. (Or the reverse order.)

With the above lookup procedure, the TLB hit time will be more than one cycle and will negate most of the benefits of reduction in the number of TLB misses. A single-cycle hit time is possible in hardware set-associative TLBs but increases the implementation cost or the TLB miss ratio:

- A dual-ported TLB can try both hash functions in a single cycle.
- The TLB can be split into two and accessed in parallel—one storing conventional entries and another storing shared entries.
- The hash function could neglect the context number and always use $h(0, VA)$, but may result in worse TLB performance on context switches.

The benefits of a hardware implementation of a common-mask set-associative TLB are not clear. However, software memory-based set-associative level-two TLBs [7] can use the two-step lookup algorithm with a minimal impact on TLB hit time and benefit from the increased TLB reach from the shared entries. The TLB hit time for a software managed TLB is already high and adding a second lookup adds only a small fraction to the TLB hit time, e.g., a superscalar processor may be able to execute the code for two probes in about the same number of cycles as a single probe.[9]

7 Common-Mask Hashed Page Tables

Page tables benefit the most from sharing of page table entries for widely shared pages, irrespective of whether the other levels of the translation hierarchy share translation entries using the common-mask scheme or any other scheme. Linear and forward-mapped page tables can do a limited amount of sharing [20]. In this section, we first show how the basic common-mask scheme described in Section 4 can be applied to hashed page tables [7]. Then we explain the benefits of sharing page table entries in any page table structure.

The common-mask scheme can be applied in a hashed page table to use a single page table entry to store the translation information for all the aliases to a physical page. We observe that a set-associative TLB is an open hash table with a fixed number of entries per set (bucket)—the associativity. A hashed page table behaves very similar to a set-associative TLB, except that we allow the hash chains to be arbitrarily long. Naturally, the common-mask scheme applies to hashed page tables exactly as described in Section 6 for set-associative TLBs.

Figure 7a illustrates a conventional hashed page table. The page table contains a set of hash buckets, where each hash bucket points to a list of nodes—-each node contains a translation for a (CTX, VPN) tuple. When the hash table is searched with VPN_{cpu} and CTX_{cpu}, the hash function $h(CTX_{cpu},VPN_{cpu})$ selects one of the hash buckets. The hash list is searched for a successful match, using the match function:

$$((CTXcpu = CTXpte) \&\& (VPNcpu = VPNpte)) \cdot$$

With the use of the common-mask scheme in the hashed page table, the new tag match function is:

$$(f(CTXcpu, CTXpte) \&\& (VPNcpu = VPNpte))$$

and the hash table lookup uses a two step algorithm: a) Index into using $h(CTX_{cpu}.id, VA_{cpu})$; b) if a) fails, index using $h(0, VA_{cpu})$.

Though the page table search requires a two-step algorithm, the page table lookup time—TLB miss

9. A TLB miss handler which looks in a memory-based set-associative TLB consists of a sequence of dependent instructions. On a superscalar processor, this may result in unused instruction issue bandwidth. Since the instructions for the second probe in the common-mask TLB are largely independent of those in the first probe, they may be able to use the wasted issue bandwidth “for free.” Note, however, that a significant portion of the time in a TLB miss handler is due to cache misses and a second probe would certainly increase the number of cache misses.

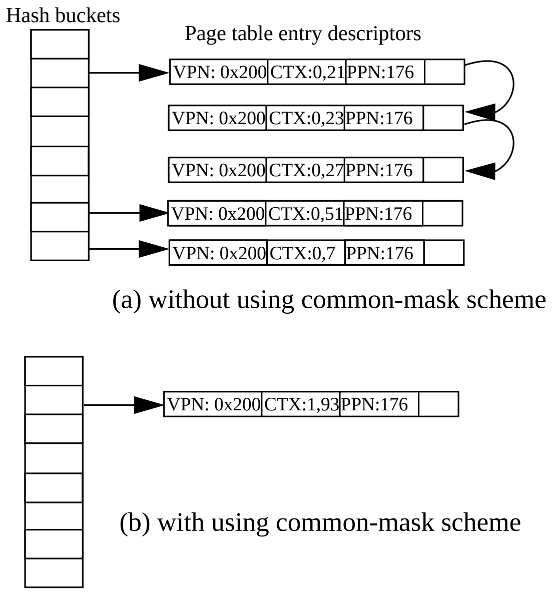
Hash buckets
Page table entry descriptors
VPN: 0x200 CTX:0,21 PPN:176
VPN: 0x200 CTX:0,23 PPN:176
VPN: 0x200 CTX:0,27 PPN:176
VPN: 0x200 CTX:0,51 PPN:176
VPN: 0x200 CTX:0,7 PPN:176
(a) without using common-mask scheme
VPN: 0x200 CTX:1,93 PPN:176
(b) with using common-mask scheme

Figure 7. Example of using scheme on page tables

An example of five mappings to the same physical page that differ in the context number only. In (a), common-masking is not used, and a separate PTE is needed for each mapping. In (b), there is only one shared PTE that has the S bit set.

penalty—is not affected much and may instead be *faster!* First, a second lookup is required only for shared translations which form a small fraction of the total number of page table lookups. Second, a software hash table lookup algorithm can be easily modified to execute two probes simultaneously, through software pipelining, to minimize the increase in the page table lookup time. Third, by sharing page table entries, the page tables occupy less memory and this reduces the average length of the hash chains and also improves the cache performance—resulting in a possibly *faster* page table lookup than in a conventional page table. If the page table lookup is slower in a common-mask page table, a level-two memory-based TLB can reduce the average miss penalty [6].

The main advantage of using the common-mask scheme in the page tables, however, is the memory savings. In the example in Figure 7, there are five different mappings to the same physical page, e.g., a text page from **/bin/csh**. The mappings are all at the same virtual address, but each is for a different process. Figure 7a shows how the mappings to the page might look like in a traditional hashed page table—one PTE per mapping. Applying the common-mask scheme results in the page table shown in figure 7b with only a single shared entry. Even increasing the number of aliases to a hundred or a thousand, the page table would continue to store a single PTE that matches with all the aliases. Reducing the amount of memory used for page tables improves system performance in many ways:

- In systems where the amount of memory dedicated for page tables is fixed, say by limiting the length of the hash bucket chains, the amount of memory mapped by the page table increases vastly by using the common-mask scheme and reduces the number of minor page faults.
- Cache pollution due to caching the PTEs is reduced as the page tables are smaller. This effect can be very significant in improving application CPU cache performance [20].
- Due to the vastly reduced length of the list of aliases attached to a physical page, alias management is much more efficient. The operating system frequently examines all aliases to a page, e.g., when clearing the modified bit after writing out a dirty page to backing store. Using the common-mask scheme, there are fewer PTE descriptors to traverse on such operations.
- The TLB miss penalty may be reduced as explained previously in this section.
- For workloads that are limited by the amount of memory in the system, the savings in page table memory usage can be used by the application to improve application performance. This is, however, not as important as the above since many performance critical applications may run on computers with sufficient memory.

There are two additional, perhaps non-obvious, advantages to using shared translation entries in any page table structure:

- The common-mask scheme reduces the number of TLB consistency operations—e.g., TLB shootdowns [2]. Consider the following example: the operating system normally maintains a reference bit per page that is used when making page replacement decisions. When the operating system clears the reference bit of a widely shared page, the reference bit is also cleared in all (possibly hundreds of) aliases' mappings. Each mapping when updated requires a TLB shootdown to invalidate the corresponding hardware TLB entries. Using the common-mask scheme, only a single page table entry needs to be updated, and when used in conjunction with a common-mask hardware TLB only a single TLB shootdown operation is required.
- Sharing of translation entries reduces the number of minor page faults serviced by the operating

system. After a process maps a shared object, the first references to the pages will result in “minor” page faults to load the private mappings into the page table for that process. These minor faults are eliminated when shared translations are used because mappings for common regions are inherited when the memory objects are mapped, and processes need not fault on first use of the shared pages. This can result in significant performance improvement in a system with many short-lived processes.

8 Operating System Considerations

The virtual memory system in the operating system must provide some support for the common-mask scheme to be useful in TLBs and page tables. It requires the addition of a policy to choose the common regions and mechanisms to maintain the common-masks, allocate virtual addresses and a hashed page table structure.

An operating system policy must choose the set of physical pages that make up the limited number of common regions. Some possible solutions are: a) A static system configuration file can identify the common regions. The file can be initialized based on profiling the common workloads for a given system. b) The system can dynamically define the common regions by keeping a count of the number of processes sharing each object and using the common-mask for the most widely shared objects. c) The system could allow user-defined common regions that may be useful in a database system.

Whichever operating system policy is used, it is important to note that the common-mask scheme is only an optimization. Conventional aliases are still supported in the page tables and the TLB. Also, dynamically changing the common regions definition is relatively easy—by clearing the correct common-mask bit in the CTX_{cpu} data structure for all the processes using the common region, the processes will demand fault their own private mappings into the page table. After unloading the shared translations from the page table and TLB, the bit can be reassigned to a new set of objects.

There are three key mechanisms that are required in the operating system to support the common-mask scheme in either the TLB and/or the page tables: a) The operating system must be able to determine if a process has mapped or unmapped any of common regions, and set/clear common-mask bits in the CTX registers accordingly (Section 8.1). b) The operating system must allocate virtual addresses for shared objects to be compatible with existing common regions (Section 8.2). c) A hashed page table gives the operating system maximum flexibility in defining the common regions (Section 8.3).

It is possible to implement sharing of translation entries in a linear or a multi-level page table [20]. Even for those page table formats, similar OS support is required. An operating system policy is required, as in the common-mask case, to choose the physical pages and address spaces whose mappings should be shared. A mechanism is required to determine if a process has mapped or unmapped any of shared regions, as in the common-mask case. Since a non-hashed page table has strict address alignment restrictions for sharing of pages, a mechanism is required to choose the right virtual addresses for mapping the shared objects. Thus, sharing of translation entries in any page table format would require similar operating system support policies and mechanisms, but the constraints under which they operate are different. For example, a common-mask system supports only a limited number of shared regions whereas a multi-level page table can support an unlimited number of shared regions; a common-mask system required shared regions to be mapped at the same virtual addresses, whereas a multi-level page table would require the virtual address to be aligned.

8.1 Maintaining the common-masks

As mentioned before, choosing the common-masks is the most important part of this scheme. In this section, we explain how the masks are maintained in CTX_{pte} and CTX_{cpu}. The operating system policy identifies *n* common regions for which it will use sharing of PTEs as explained before. One bit in the common-mask bit vector is allocated to each common region.

When the operating system loads a translation entry into the page table and/or TLB, it specifies CTX_{pte} for the translation. Normally, the translation is tagged with the context id of the process to which the translation belongs (CTX_{pte}) and the virtual address (VPN_{pte}) for which this translation corresponds. In the common-mask scheme, this remains unchanged except for translations to pages within a common region where CTX_{pte} specifies a bit vector of the common regions to which this shared translation belongs.

When a process maps an object, **mmap**, the operating system checks to see if this process maps any of the *n* common regions correctly—at the correct virtual addresses and with the right protections. If so, the operating system will set a bit in the bit vector of CTX_{cpu} in its data structures—and in the hardware register if the process is currently running. As mentioned before in Section 7, it is important to note that the process' mappings are ready in the page table "for free," and the process does not have to incur minor page faults to initialize them.

When a process unmaps an object, **munmap**, the operating system checks to see if it belongs to a common region that this process is sharing. If so, then it simply clears the bit in the bit vector, CTX_{cpu}. There are two very important benefits from this action. First, the translations need not be removed from the page table or invalidated from the TLB. They need to be removed from the page table and TLBs only when the shared region is discarded, which is infrequent. This is a significant benefit since TLB invalidates, and page table deletions in a multiprocessor system are expensive. Second, the current process does not restrict or interfere with the use of the common region by any other process in the system. If a process unmaps an object that is part of a common region, other processes continue to use that common region unhindered. Global bits do not have this feature—processes cannot unmap objects from virtual addresses for which other processes use the global bit.

8.2 Choosing virtual addresses

Programs mapping objects that belong to common regions must preferentially be allocated at one (or a few) special address(es)—each object in a common region has one or more virtual addresses for which aliases can use the shared entries. One simple solution is to maintain a preferred address for each object, and let the operating system automatically attempt to map objects at their preferred addresses in all processes. Note that, in practice, almost all user programs allow the operating system to pick the virtual address at which an object is mapped. With 64-bit address spaces, an operating system will nearly always be able to choose unique addresses for objects, but may use the virtual address space in a sparse fashion.

Programs can still request shared objects to be mapped at specific virtual addresses but may be unable to benefit from the common-mask TLBs and page tables when sharing the objects with other processes. They may instead be given process-private mappings (they still share the same physical memory for the objects), or may be allocated a new common region if enough programs use the same non-standard address.

8.3 Choosing the page table structure

Hashed page tables are preferred when implementing sharing of translations because they do not place any restrictions on the size or alignment of the shared translations, and any arbitrary page or collection of pages can be shared. Though sharing of translations can be implemented in linear and forward-mapped page tables [20], the sharing can only occur at the granularity of a page table (e.g., all of a 256K or 4MB address space has to be shared). Such size and alignment restrictions reduce the effectiveness of page table sharing, e.g., shared libraries typically have some private data that is not shared across processes (e.g., the global offset table in ELF objects [13]) but is mapped at a fixed offset from the text segment. This makes it difficult to find a contiguous, aligned chunk of virtual address space that is shared identically with another process. Hashed page tables do not place any such restrictions and the operating system can share more pages.

More importantly, for multi-level and linear page tables, the virtual addresses allocation for the shared objects results in a sparsely populated address space. For example, to share translations in a linear page table, each shared library would require two consecutive, aligned 4MB regions of virtual address space—one for the text and another for the data. Libraries being small, only a small fraction of each 4MB is used and the rest is reserved. Linear and forward-mapped page tables are inefficient at supporting such sparse address spaces making hashed page tables more attractive.

We would like to emphasize, however, that while hashed page tables have some advantages in supporting sharing of page table entries, the common-mask hardware and level-two TLBs can be used irrespective of the page table structure and irrespective of whether page entries are shared. Thus, a common-mask level-two TLB can be used together with a forward-mapped page table that shares translations.

9 Performance Evaluation

In this section, we discuss the workloads, methodology, metrics, and results of the performance evaluation of the benefits of using the common-mask scheme in hardware TLBs, level-two TLBs, and page tables.

9.1 Workloads

We use the **AimIII** and **Kenbus** [12] standard multi-user benchmarks, in addition to several benchmarks constructed by running multiple copies of the some **SPEC** [12] programs.[10] We used a uniprocessor SPARCserver 10 with 96MB of main memory for all our TLB simulations

All the workloads are examples of read-only sharing of text and library segments. Database programs use read / write heap sharing and would benefit from the common-mask TLB. We use the Oracle Financials database system run on a 8-processor SPARCserver with 384MB of main memory. We are unable to report TLB simulation numbers for databases as we could not get access to a commercial database server that would run our modified operating system.

9.2 Methodology

We evaluate the performance of a common-mask TLB using trap-driven simulation [17] implemented in **foxtrot** [15]—a Solaris® 2.1 based operating system that counts the number of user TLB misses for a workload. Our simulation environment does not include kernel TLB misses, but includes the effect of context switches in the multi-programmed workloads. Kernel TLB misses will certainly influence system performance, but we cannot guess whether it will improve or worsen our results without measuring them.

Our operating system does not include the mechanisms and policies needed to select common regions or manage common-masks and virtual address allocation. We instead assume in our simulation that an infinite number of common regions are available, and that all aliases could use the common-mask scheme. Table 5 shows that most workloads fit within sixteen statically determined common regions, and assuming an infinite number of common regions gives no additional advantage. In a real system, the common regions may have to be dynamically adjusted to the workload and may not exploit all available sharing.

We measure the page table memory demand of the workloads by taking a snapshot of the state of the page tables and counting the number of valid PTEs. The workloads used in this study do not manipulate their address space aggressively, and its structure does not change over the life of the workload. To estimate the page table memory demands for the common-mask hashed page tables, we define a set of common regions for each workload, and use a single page table entry for each physical page in the common region. With sixteen common regions, we were able to eliminate all aliases for these workloads. A typical UNIX® system has many daemons running and the workloads map the common regions used by these daemons for shared libraries. For example, a common region consisting of **libc.so.1**, **libdl.so.1** and **ld.so** text and data segments is used by nearly *every* process.

10. The SPEC programs were compiled to use dynamic linking—SPEC benchmarks are normally linked statically but most “real-world” programs on Solaris use dynamic linking to libraries.

9.3 Metrics

We use the percent reduction in the number of user TLB misses as the metric for evaluating the performance of a common-mask TLB compared to a conventional single-page-size TLB with the same number of entries, associativity, and replacement policy, i.e.,

$$\left(\frac{missesold - missesnew}{missesold}\right) \times 100\%$$

Appendix A includes the absolute number of TLB misses for the conventional TLBs.

TLB misses are only one part of the system performance, and TLB miss penalty is equally important. Our simulation environment does not measure TLB miss penalty, but a *critical miss penalty* can be estimated from Tables 6 and 3 as:

$$OriginalTLBmissPenalty \times \left(\frac{100}{100 - r}\right)$$

where r is the percent reduction in the number of TLB misses. The critical miss penalty is the TLB miss penalty for the common-mask page table that exactly compensates for the reduction in the number of TLB misses. If the common-mask TLB has 25% fewer TLB misses, the critical miss penalty is 1.33 times the original TLB miss penalty. The TLB miss handler may require two traversals of a common-mask page table for the shared mappings and the TLB miss penalty is:

$$FirstScanPenalty + (f \times SecondScanPenalty)$$

where f is the fraction of TLB misses to the shared pages. *SecondScanPenalty* is smaller than FirstScanPenalty as it does not include costs such as trap entry. *FirstScanPenalty* may be smaller than the Original TLB miss penalty as the page tables are smaller, resulting in shorter search times and better cache behavior.

The metric we use to evaluate the memory savings from using a common-mask hashed page table is the percentage reduction in the number of page table entries needed to store the mappings used by the workload. We do not include in our calculation the extra storage required to maintain the common region data structure—a list of processes mapping the common regions—which is O(number of processes) sharing a common region.

9.4 Common-mask TLB performance

Tables 2 and 3 show the percent reduction in number of user TLB misses for 64-,128- and 256-entry fully-associative TLBs; 128- and 256-entry 4-way set-associative TLBs using LRU [4]; and RANDOM replacement, respectively. There are two trends that the numbers illustrate.

Workload	64-entry FA	128-entry FA	256-entry FA	128-entry 4-way SA	256-entry 4-way SA
aimIII (100 users)	21.0%	31.4%	40.1%	30.1%	39.4%
kenbus (80 users)	-0.3%[a]	12.0%	30.9%	13.2%	31.3%
kenbus (50 users)	1.3%	14.8%	32.0%	12.8%	32.9%
doduc (10 copies)	0.4%	35.0%	48.1%	31.7%	49.5%
mdljdp2 (10 copies)	-1.8%[a]	7.7%	19.8%	8.0%	23.9%
ora (10 copies)	21.8%	22.4%	32.4%	23.8%	34.9%
ear (10 copies)	38.9%	43.1%	56.0%	50.1%	50.3%
mdljsp2 (10 copies)	5.4%	10.1%	30.9%	20.9%	33.4%
fpppp (10 copies)	4.5%	20.6%	41.1%	74.3%	41.3%
gcc (10 copies)	-0.1%[a]	1.7%	20.9%	3.1%	22.0%

Table 2. Percent Reduction in user TLB misses with common-mask TLB (Random replacement)

a. Our common-mask TLB simulator executes slower than our conventional TLB simulator resulting in taking more CPU quanta to execute. This results in a small extra number of misses in a common-mask TLB.

Workload	64-entry FA	128-entry FA	256-entry FA	128-entry 4-way SA	256-entry 4-way SA
aimIII (100 users)	32.8%	38.1%	49.9%	38.4%	46.4%
kenbus (80 users)	2.6%	23.4%	45.0%	20.8%	40.6%
kenbus (50 users)	3.7%	23.5%	45.5%	21.5%	53.7%
doduc (10 copies)	21.5%	42.4%	53.4%	41.6%	53.9%
mdljdp2 (10 copies)	10.0%	18.5%	22.7%	18.6%	22.7%
ora (10 copies)	26.7%	40.0%	51.0%	43.8%	49.2%
ear (10 copies)	31.4%	45.1%	72.3%	45.0%	64.8%
mdljsp2 (10 copies)	5.7%	21.2%	31.8%	21.8%	35.4%
fpppp (10 copies)	17.5%	32.9%	40.8%	76.9%	44.5%
gcc (10 copies)	0.0%	3.4%	31.9%	12.9%	27.8%

Table 3. Percent Reduction in user TLB misses with common-mask TLB (LRU replacement)

First, the common-mask scheme is more effective with larger TLBs. The 256-entry TLBs are able to hold most of the working set for a process and a most TLB misses are due to context switches, whereas the 64-entry TLBs incur many capacity misses with even a single process. The common-mask TLB shares some TLB entries across all the processes, and significantly reduces the number of TLB misses per context switch in the large TLBs. In the smaller TLBs, capacity misses replace some sharable TLB entries and result in a smaller decrease. The performance of **kenbus** and **gcc** clearly illustrates this effect of capacity misses.

Second, the replacement policy used is important. The LRU replacement policy is less likely to replace widely shared TLB entries as it uses the reference history across all processes to make replacement decisions. Since RANDOM does not distinguish shared TLB entries, it is likely to replace even widely shared TLB entries. This reduces the effectiveness of the common-mask TLB—Table 3 shows a smaller performance improvement than in Table 6. The absolute number of TLB misses is also much higher when using the RANDOM replacement policy (Appendix A).

Workload	4096-entry direct-mapped	4096-entry 2-way SA (RANDOM)
aimIII (100 users)	44%	53.1%
kenbus (80 users)	38.0%	40.1%
kenbus (50 users)	46.9%	62.6%
gcc (10 copies)	31.4%	45.0%

Table 4. Percent Reduction in user TLB misses with common-mask level 2 TLBs

Table 4 shows the percent reduction in number of user TLB misses in a level-two 4096-entry TLB (direct-mapped and 2-way set-associative). The level one TLB is a conventional 64-entry fully-associative TLB with LRU replacement. Multi-level inclusion [1] is not maintained. The common-mask scheme is very effective in level two TLBs with 30-60% reduction in the number of TLB misses. The TLB miss penalty for a level two TLB miss can be quite significant—a page fault—which makes the reduction in number of TLB misses important.

Therefore, we conclude that there are two conditions to be met for the common-mask scheme to be most effective in TLBs: a) There must be sharing of physical objects between the different programs of the workload. Most programs share library code and databases share the buffer cache mappings; b) The TLB reach must be large enough to be able to map a single program's, not workload's, working set[11] size. Recent TLB architectures are increasing the hardware TLB reach through use of superpages and subblocking, decreasing capacity misses and making it more attractive to implement the common-mask scheme.

9.5 Common-mask page table evaluation

The common-mask scheme can be used to reduce the amount of memory used for page tables by combining the storage for multiple aliases into a single PTE. Reducing the amount of memory improves system performance in many ways as described in Section 7.

11. The working set size used in the context of TLB performance assumes T, the working set parameter, to be equal to the CPU time quanta. The working set size in a traditional operating system sense uses a much larger T. Thus, the working set size here is the number of distinct pages referenced between context switches and is much smaller than the program working set size resident in main memory.

Workload	#PTEs in a Hashed Page Table	#PTEs in a Common-mask Hashed Page Table	% reduction in number of PTEs	Number of common regions used
aimIII (100 users)	5555	1455	74%	1
aimIII (500 users)	27555	7055	75%	1
kenbus (50 users)	21800	1493	93%	7
kenbus (80 users)	34880	2363	93%	7
Idle Workstation	13388	3521	74%	16
Oracle Financials	1895155	57862	97%	9
Oracle Financials (64MB superpage)	338675	41478	88%	9

Table 5. Memory Savings using the common-mask Scheme for Hashed Page Tables

Table 4 shows the number of page table entries in a page table with no sharing, the number of PTEs in a page table using the common-mask scheme and the percent reduction by using the common scheme. The fourth column shows the number of common regions used by this workload. The table shows a significant reduction in the number of PTEs that the page table needs to store, up to 97%.

The Oracle Financials database workload includes a 64MB buffer cache shared by 95 processes running on an 8-processor system—commercial workloads often use a larger buffer cache with more server processes. Using separate page tables for each process, results in about 16MB of memory for the page tables, a 25% overhead. While memory utilization is not an issue in database servers, the 16MB of page tables cause a significant amount of cache pollution and degrade system performance. A recent study [20] shows a 10% speedup in the TPC-B benchmark through use of page table sharing. The common-mask scheme allows a single copy of the page table entries for the buffer cache and code pages resulting in much smaller page tables.

One way to address the database buffer cache page table problem is to use 64MB superpage mappings [15]. We would then use a *single* PTE per process for the buffer cache and improve TLB performance as well. The workload, using a superpage mapping for the buffer cache, still shows a significant amount of read-sharing among the processes and the common-mask scheme continues to help significantly in reducing the page table size. The data base programs use large text segments, 6MB and 2.5MB, and several shared libraries that can be mapped by the common regions.

Since a common-mask system supports only a limited number of common regions, the number of common regions used for a workload is important. AimIII requires only a single common region for its text segment. Kenbus uses four common regions for each of four widely used process' text segments and three "system" common regions that are already used by common daemons such as the router and automounter. The Oracle Financials uses three common regions, two program text and one buffer cache, and six system common regions. The idle workstation was running the X window system and defines most of the system common regions. Even on an idle system, the amount of memory needed for the page tables is reduced by 74%.

10 The Common-mask Scheme in Other Domains

This paper introduces the common-mask scheme in the context of address translation to share translation entries for aliases. However, the scheme we propose here is quite general and may be applicable in other domains where multiple keys map to the same data, i.e., aliases. In this section, we illustrate the use of the common-mask scheme in a virtually-tagged cache and a relational database. Note, however, these only illustrate that the scheme may be used in other domains and we have not studied the viability of practical implementations in these domains.

10.1 Common-mask scheme in virtually-tagged caches

The common-mask scheme can be applied nearly identically to virtually tagged[12] caches that include the context number in the cache tag. Virtually-tagged caches have similar problems with multiple aliases to heavily shared objects, where the cache cannot use cache lines that belong to another process even though they may be referring to the same data. This problem can be solved by matching the context number using the common-mask scheme, with the operating system utilizing the same common regions as the ones used for TLBs and page tables. However, since virtually-tagged caches are usually set-associative, and are speed-critical, indexing using the multiple hash probe as described in Section 6 may result in an impractical design. Note that a system could employ the common-mask scheme in the TLB, while the caches can continue to use conventional context number matching. In other words, though processes can share TLB entries, they do not share cache lines, and the cache behaves as if there was no common-masking.

12. There is no duplication of aliases in physical-tagged caches.

10.2 Common-mask scheme in relational databases

A relational database is a simple table with each row containing a set of attribute fields—some of which form a key. Many databases have multiple keys associated with the same attribute values. To reduce duplication and save on storage, database administrators design a *database schema* by splitting the single large table into multiple tables replacing the common attribute columns with pointers to a smaller relation. During query processing, the large table (or a subset) is reconstructed using a JOIN operation.

For example, an airplane part inventory relation storing the details of the parts used to custom-build an airplane could consist of the following schema: Serial#, Part#, Part Description(long). Each row in the relation would contain the description of one of the thousands of parts used. Though the planes are custom built, many planes (serial numbers) would use the same part—e.g., (Plane1,Part123) and (Plane2,Part123) may have the same part descriptions. A better schema would be: Serial#, Part#, PartID in relation1 and PartID,Part Description in relation2. The new schema uses much less storage by eliminating the duplicate part descriptions of the first schema.

Using the common-mask scheme, the keys can be extended and assigned values such that multiple key values will match the key stored in a single row of the database. This eliminates rows in the relation that store duplicate information associated with different keys. Hence, instead of using two relations in the schema, a single relation may be used.

In the above example, we could use the common-mask scheme for assigning the Serial# and Part# for the planes and use the single relation schema but still get the benefit of the reduced storage as in the two-relation schema. First, divide the parts into common groups that are often used together in an airplane. Second, split the Serial# assigned to airplanes into two fields—a bit vector identifying the common part groups that were used and a private unique id (possibly the serial# in the original schema). Now, each row of the relation either contains the description of a part used in a single plane—the serial# stores the private id—or contains the description of a part used in multiple planes—the serial# stores a bit vector in the relation.

There are two costs associated with this scheme. First, the keys must be allocated carefully to maximize the sharing of the database rows. The problem is similar to the allocation of the common-mask bits in address translation and it may not be possible to eliminate *all* duplicate rows as the key will have a limited number of bits. Second, the normal index structures and lookup routines need to be modified to account for the special multi-valued keys. Relational databases sometimes use a hashed index structure and the common-mask scheme is directly applicable in the hash index also. The common-mask scheme can be used in any hash table which maps multiple keys to the same data, for example, a set-associative TLB and a hashed page table. The hash table, however, needs multiple probes which can be expensive.

Another domain where the common-mask scheme may be used is in associative processing and content addressable memories. Associative processors used in applications such as natural language processing and image recognition are typically based on fully-associative memories and deal with many aliases. It is possible that the common-mask scheme, perhaps with a different domain-specific match function, is applicable in these domains.

In Section 3, we propose the generic approach to sharing of translation entries, but did not solve the generic problem as we considered it impractical for our application. Future work could include finding a more efficient solution for the generic problem, and finding a solution for relatively static domains.

11 Conclusions

Many operating systems allow multiple processes to share the same physical object. This introduces aliases into the system where different virtual addresses map to the same physical address. Aliases increase the size of the page tables, exerting increased pressure on TLBs and virtually-tagged caches. Use of segments and a global address space model is one approach to avoiding aliases. Most UNIX implementations, however, use a private address space model and allow for aliases.

In this paper, we present a *common-mask* scheme that allows multiple translations from different processes to the same physical address to share a single translation entry in a TLB or page table. The scheme allows for a limited number of *common regions* that processes can share. A common region is a set of physical pages that a process maps using a fixed virtual address and a fixed set of permissions for each physical page in the set. The process context id is extended with a bit vector that identifies a set of common regions that a process shares with other processes. TLB and page table entries support shared entries by storing a bit vector that can identify common regions instead of a single process context id. The scheme requires aliases that use the common region to use the same virtual address, and can only support a few common regions. Further, the common-mask TLBs and page tables continue to work with conventional private mappings exactly as in the original system.

We show how the common-mask scheme can be applied to all levels of the translation hierarchy—hardware TLBs (fully-associative and set-associative), memory-based TLBs and page tables. The scheme we propose is quite general and can be applied in domains other than address translation where multiple keys map to the same data e.g., virtually-tagged caches, associative processors [26], relational databases and any hash table.

We show that the use of the common-mask scheme in hardware TLBs can substantially reduce the number of TLB misses in a multi-user workload. We show a 50% reduction in the number of TLB misses for a 256-entry fully-associative TLB. However, the scheme does not work very well in TLBs with a small TLB reach. We expect the common-mask scheme to be attractive when combined with other techniques that improve TLB reach—superpages, subblocking, and larger TLBs. A set-associative common-mask TLB would require accessing two sets and may increase TLB hit time due to the lack of an unique index function for both the shared and private translations and the benefits of a hardware set-associative implementation are not clear.

In software-managed memory-based set-associative TLBs, the common-mask scheme also requires accessing two sets. However, the increase in hit time is small as the TLB hit time for a conventional software-managed TLB is already high and the second probe adds very little to the hit time.

The most attractive use of the common-mask scheme is in hashed page tables, where it can dramatically reduce the amount of memory required to store the page tables. Using the common-mask scheme reduces the size of the page tables by 97% for a commercial database server. The smaller page tables have many indirect benefits—reducing the TLB miss penalty, reducing cache pollution due to the page tables, reducing the number of minor page faults and, in conjunction with a common-mask hardware TLB, reducing the number of TLB shootdowns.

Acknowledgments

We would like to acknowledge Basem Nayfeh and Allan Packer of Sun Microsystems for their help in obtaining performance numbers for this paper, and Mark Hill of University of Wisconsin-Madison for suggesting the name, "common-mask".

References

[1] J. Baer and W. Wang, "On the Inclusion Properties of Multi-level Cache Hierarchies", *15th Annual International Symposium on Computer Architecture*, June 1988, pp. 73-80.

[2] D. Black *et al.*, "Translation Look-Aside Buffer Consistency: A Software Approach", *3rd Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III),* April 1989, pp. 113-122.

[3] J. Chase, H. Levy, M. Feeley and E. Lazowska, "Sharing and protection in a single address space operating system", Technical Report 93-04-02. University of Washington, Dept. of Computer Science and Engineering, April 1993.

[4] Y. Deville and J. Gobert, "A Class of Replacement Policies for medium and high-associativity structures", *Computer Architecture News*, 20(1), Mar 1992, 55-64.

[5] R. Gingell *et al.*, "Shared Libraries in SunOS", *Proceedings of the Summer USENIX Conference,* Summer 1987.

[6] J. Hennessey and D. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 1990.

[7] J. Huck and J. Hays, "Architectural Support for Translation Table Management in Large Address Space Machines", *20th Annual International Symposium on Computer Architecture*, May 1993, pp. 39-50.

[8] G. Kane, "MIPS RISC Architecture", Prentice Hall, 1989.

[9] M. Milenkovic, "Microprocessor Memory Management Units", *IEEE Micro*, vol. 10, no. 2, April 1990, pp. 70-85.

[10] E. Koldinger, J. Chase, and S. Eggers, "Architectural Support for Single Address Space Operating Systems," *5th Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V),* October 1992, 175-186.

[11] *SPARC Architecture Manual*, Version 8, SPARC International, Menlo Park, CA, 1991.

[12] *SPEC Newsletter,* vol. 3 (2), Spring 1991.

[13] *System V Application Binary Interface*, USL, Prentice-Hall, 1992.

[14] M. Talluri, S. Kong, M. Hill, and D. Patterson, "Tradeoffs in Supporting Two Page Sizes", *19th Annual International Symposium on Computer Architecture*, May 1992, pp. 355-363.

[15] M. Talluri and M. Hill, "Surpassing the TLB Performance of Superpages with Less Operating System Support", *6th Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI),* October 1994, pp 171-182.

[16] J. Torrellas *et al.*, "Characterizing the Caching and Synchronization Performance of a Multiprocessor Operating System", *5th Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V),* October 1992, 162-174.

[17] R. Uhlig, D. Nagle, T. Mudge, and S. Sechrest, "Trap-driven Simulation with Tapeworm II", *6th Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI),* October 1994, pp. 132-144.

[18] B. Wheeler and B. Bershad, "Consistency Management for Virtually Indexed Caches", *5th Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V),* October 1992, pp 124-136.

[19] S. E. Wilton and N. Jouppi, "An Enhanced Access and Cycle Time Model for On-Chip Caches", WRL Research Report 93/5, 1993.

[20] H. Yoo and T. Rogers, "UNIX Kernel Support for OLTP Performance," *1993 Winter USENIX*, January 1993, pp. 241-247.

[21] Douglas W. Clark and Joel S. Emer, "Performance of the VAX 11/780 Translation Buffer: Simulation and Measurement", *ACM Transactions on Computer Systems,* February 1985, pp. 31-62

[22] R.O. Simpson, P.D. Hester, "IBM RT PC ROMP Processor and Memory Management Unit Architecture", *IBM System Journal*, January 1987, pp. 346-360.

[23] Peter J. Denning, "The Working Set Model for Program Behavior", *Communications of the ACM,* May 1968, pp. 323-333.

[24] Peter J. Denning, "Virtual Memory", *Computing Surveys*, September 1970, pp. 153-189.

[25] E. J. Organick, "The Multics System: An Examination of its Structure", MIT Press, 1972.

[26] IEEE Micro, special issue, "Associative Memories and Processors", Parts 1 and 2, June and December 1992.

[27] Richard F. Rashid *et al*, "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures", *IEEE Transactions on Computers*, August 1988, pp. 896-908.

[28] Bradley Chen, Anita Borg and Norman Jouppi, "A Simulation Based Study of TLB Performance", *19th Annual International Symposium on Computer Architecture*, May 1992, pp. 114-123.

[29] David Nagle, Richard Uhlig and Tim Stanley, "Design Tradeoffs for Software Managed TLBs", *20th Annual International Symposium on Computer Architecture*, May 1993, 27-38.

[30] Ruby B. Lee, "Precision Architecture", *IEEE Computer*, January 1989, 78-91.

[31] Ed Silha, "The PowerPC Architecture, IBM RISC/System 6000 Technology, Volume II", IBM Corporation, 1993.

[32] Jochen Liedtke, “A High Resolution MMU for the Realization of Huge Fine-Grained Address Spaces and User Level Mapping”, German National Research Center for Computer Science Technical Report No. 829, March 1994.

[33] Jochen Liedtke, “A Virtually Indexed Cache memory with Efficient Synonym Handling”, German National Research Center for Computer Science Technical Report No. 791, October 1993.

Appendix A

Workload	**64-entry FA**	**128-entry FA**	**256-entry FA**	**128-entry 4-way SA**	**256-entry 4-way SA**
aimIII (100 users)	32.8%	38.1%	49.9%	38.4%	46.4%
kenbus (80 users)	2.6%	23.4%	45.0%	20.8%	40.6%
kenbus (50 users)	3.7%	23.5%	45.5%	21.5%	53.7%
doduc (10 copies)	21.5%	42.4%	53.4%	41.6%	53.9%
mdljdp2 (10 copies)	10.0%	18.5%	22.7%	18.6%	22.7%
ora (10 copies)	26.7%	40.0%	51.0%	43.8%	49.2%
ear (10 copies)	31.4%	45.1%	72.3%	45.0%	64.8%
mdljsp2 (10 copies)	5.7%	21.2%	31.8%	21.8%	35.4%
fpppp (10 copies)	17.5%	32.9%	40.8%	76.9%	44.5%
gcc (10 copies)	0.0%	3.4%	31.9%	12.9%	27.8%

Table 6. Percent Reduction in user TLB misses with common-mask TLB (LRU replacement)

About the authors

Yousef A. Khalidi is currently a Senior Staff Engineer and Principal Investigator at Sun Microsystems Laboratories. His interests include operating systems, distributed object-oriented software, computer architecture, and high-speed networking. He is one of the principal designers of the Spring operating system, and a co-winner of Sun's Presidential Award in 1993. He has a Ph.D. in Information and Computer Science from Georgia Institute of Technology, where he was one of the principal designers of the Ra and Clouds operating systems.

Madhusudhan Talluri is a Staff Engineer at Sun Microsystems Laboratories. His interests include computer architecture, operating systems, and virtual memory. He has worked on the support of superpages in the UltraSPARC TLB, page tables, and operating system. He earned a BTech degree in Computer Science and Engineering from the Indian Institute of Technology, Madras, India, in 1989, and M.S. and Ph.D. degrees in Computer Science from the University of Wisconsin, Madison, in 1991 and 1995, respectively. He is a member of ACM, IEEE, and IEEE Computer Society.