Architectural Support for Translation Table Management in Large Address Space Machines

by

Jerry Huck
Hewlett Packard
19410 Homestead Ave.
Cupertino, CA 95014

Jim Hays
EcoSystems Software, Inc.
10055 Miller Ave., Suite 201
Cupertino, CA 95014

Abstract

Virtual memory page translation tables provide mappings from virtual to physical addresses. When the hardware controlled Translation Lookaside Buffers (TLBs) do not contain a translation, these tables provide the translation. Approaches to the structure and management of these tables vary from full hardware implementations to complete software based algorithms.

The size of the virtual address space used by processes is rapidly growing beyond 32 bits of address. As the utilized address space increases, new problems and issues surface. Traditional methods for managing the page translation tables are inappropriate for large address space architectures.

The Hashed Page Table (HPT), described here, provides a very fast and space efficient translation table that reduces overhead by splitting TLB management responsibilities between hardware and software. Measurements demonstrate its applicability to a diverse range of operating systems and workloads and, in particular, to large virtual address space machines. In simulations of over 4 billion instructions, improvements of 5 to 10% were observed.

1. Introduction

Virtual memory, *VM*, is a fundamental abstraction of storage used by computer systems to support concurrent execution of processes. Processes can be protected from other processes execution and processes can view storage in a simplified, uniform manner.

Virtual memory defines a mapping function from one address space to some other address space. Traditionally, that mapping is a single translation from a virtual address, local to the process, to a physical address that directly accesses storage. These mappings are termed *translations*. The instruction set provides management instructions to enable and disable the translations, change translations, and control the protection model that is often associated with the translation mechanism.

Virtual memory translation is often specified by the architecture in terms of a memory-based table with the expectation that some intermediate storage element will hold a subset of these translations. The translation lookaside buffer (TLB) is the most common structure used to hold this subset. The processor interrogates the TLB with a virtual address and searches for the corresponding translation. If found, the hardware uses the translation to validate the access and locate the data. Many approaches to the design of TLBs have been implemented and measurements taken of their behavior [Clar85][Tayl90].

If the TLB does not contain the translation (an event known as a *TLB miss*), then typically some type of memory based table, the *page table*, is accessed and the translation is entered into the TLB.

Besides the translation, the page table entry usually holds protection information that controls access (read/write/execute) and status bits. The status bits might record if the page has been recently referenced (reference bit) and if the page has been written (dirty bit). The operating system may store additional information in the page table not needed by the TLB, status bits, or links to other software tables.

Overall performance of a computer system is dependent on TLB access and management overhead. Measurements of large scale applications, databases, networking, and operation systems behavior indicate that a significant number of the CPU cycles can be consumed in TLB management. Later sections will better quantify the costs; large scale data-base intensive applications incur 5-18% overheads. Extreme cases show greater than 40% TLB overhead.

The page table structure of any VM system attempts to optimize three different characteristics:

1. Minimize the time to service a TLB miss.
2. Minimize the physical memory space to maintain the translations for the currently mapped pages.
3. Maximize the flexibility for software to support a variety of VM mechanisms and capabilities.

Computer systems support the page table structure using hardware, or hardware with some assistance by software, or entirely by software. Hardware approaches seek the highest performance while software only mechanisms retain the flexibility to easily adapt the page table structures to changing requirements.

No practical organization optimizes this set of characteristics for all types of workloads. Organizations effective for one workload may be a poor match for another. Different operating systems make different demands on the translation mechanisms.

The trends in computing point to changes in the utilization of the address space. Object-oriented systems, mapped files, shared objects, and distributed computing all increase the size of the address space used by a process, encourage more sharing and decrease the locality of the resulting virtual memory address stream. The translation structure's performance is influenced by these changes. Later measurements quantify the very different behavior of simple program and more complex operating system execution.

Independent of the particular organization, all page tables are simply a data structure that is primarily designed for efficient retrieval of a translation using the virtual address as a search key. Searching is a large field and well researched field.

0884-7495/93 $3.00 © 1993 IEEE

Many data structures and search techniques are possible. Choosing a high performance implementation is difficult. The addition of just a single extra memory reference may be very costly. Each clock cycle of the TLB miss handler must be carefully considered and measured. For example, some page structures use a hash table for searching. Theory suggests that hash tables should be a prime number in size. Practice dictates that these tables are all powers of 2 in size.

The following section describes the two common page table structures and analyzes their characteristics. This is followed by the proposed alternative. The final section quantifies the performance of these structures over a range of workloads and operating systems.

2. Existing Virtual Memory Architectures

Two styles of page table organization dominate virtual memory architectures today: *forward-mapped* and *inverted*. Forward-mapped page tables are the most common structure used for 32-bit or less virtual memory architectures. Inverted, or alternatively *reverse-mapped*, page tables - IPTs - have been typically used by large address space architectures.

2.1 Forward-mapped page tables

Forward-mapped or alternatively *multi-level* page tables generally use bits out of the virtual address to index a hierarchy of tables. The final level of the hierarchy, the leaf, contains a validity indicator, the physical page number, and any status or protection bits. For a particular virtual address, there is one single location that holds the translation. Most architectures with this structure allow portions of the hierarchy to be unallocated by using validity bits in the higher levels. Some architectures provide a short circuit approach that promotes leaf pages to a higher place in the hierarchy when the address space is sparsely used. Figure 1 shows an example table illustrating this mechanism. The root pointer is used to start the search. Each index merges bits from the entry with more of the virtual address bits. The complete physical address is formed with the page offset bits in the virtual address and physical page number in the leaf page.

The forward-mapped table is generally a per-process table. Some control register holds a pointer to the first level of the table. The amount of storage being used for the page tables is a function of the amount of allocated virtual memory. Portions of the table need not be resident if none of its entries are mapped.

The page table itself can be referenced virtually or physically. If physically referenced, then some table manager controls the amount of physical memory being used.

As an example, to map 32-bits of address space with 4Kbyte pages, a system might allocate 1024 entries in the root table. Each root table entry in turn points to leaf pages each containing 1024 entries.

Since these per process page tables map the same virtual address values, the virtual address is often augmented in the TLB with an address space identifier (ASID) or process identifier. The ASID acts to form a unique global address for each process and avoids the need to purge the TLB on context switch by preventing erroneous matching of a virtual address from one process with a translation owned by another process.

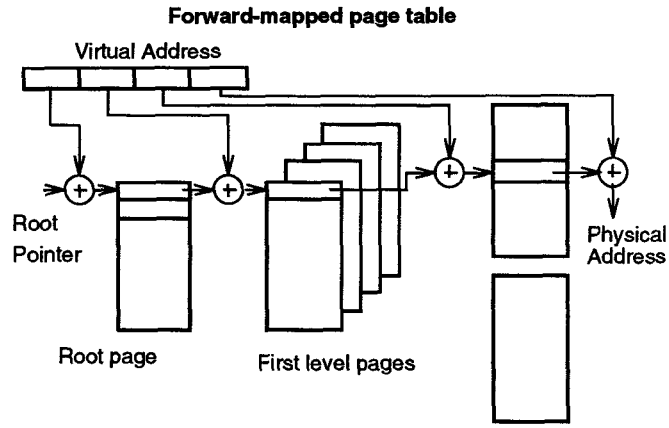


Figure 1

By duplicating entries on a per process basis, forward-mapped page tables offer a very flexible VM structure to the operating system. It has full support for aliasing, copy-on-write, and independent protection views (different protection mode for different virtual addresses).

Time: Servicing a TLB miss requires the loading of pointers from each of the upper levels of the hierarchy and finally the loading the last entry. The total cost to service a miss involves:

CPU overhead to suspend execution and step
through each part of the algorithm,
Memory and/or cache references to each level,
Possible TLB miss with virtual tables,
Possible page fault on the table itself, and
Possible updates of the dirty and reference bits.

In the earlier example, a TLB miss requires 2 memory references and perhaps additional TLB misses if the page table itself is virtually referenced.

Minimizing the miss time requires careful design of the system. Some multiprocessing systems require bypassing the cache for page structure references to avoid synchronization problems with table management [Appo88]. Even on systems that use the cache, relatively high miss rates occur. Later measurements quantify these values.

The nature of the TLB miss address stream is an important determinant of the system's performance. Largely sequential or densely packed TLB miss addresses match the characteristics of forward-mapped page tables. On the other hand, more sparse and distributed TLB miss addresses can result in longer miss service times.

An interesting variant on the forward-mapped table is a single flat table that is indexed by the virtual page number [DEC83]. The table is very large but only the used pages need to be allocated. From a time standpoint, only one memory reference and potentially one nested TLB miss is possible. For sequential miss patterns the extra TLB miss only occurs on page crossings on the table itself. For sparse accesses, this can require both an expensive memory reference and an expensive nested TLB miss.

In general, the introduction of just a single additional cache miss in TLB miss handling can greatly reduce performance. With cache miss times increasing from 10-20 cycles toward

30-70 cycles, this time could easily double or triple typical TLB miss times. Later measurements quantify some of these effects.

One additional time issue is the hit rate of the TLB itself. Forward-mapped tables generally use address aliasing to share data and require multiple entries in the TLB for the shared pages. This effect reduces the apparent size of the TLB vs a system that uses a single entry to map all access to a shared page. An even more costly approach, requires the purging of translations between every context switch. Programming trends to access mapped files, and shared memory objects will increase the occurrence of this sharing. Forward-mapped systems sometimes allow global sharing with some restrictions on the allowed protection model. For example, the VAX architecture allows all processes to share an address in system address space in the same way, say, read-only.

Space: The space required for page tables is a function of the amount and distribution of allocated virtual memory. In the best case, all entries of the leaf pages are used. In the example, this implies 1 word of overhead for every mapped page (4 bytes/4K bytes ≈ .1%). In the worst case, sparse allocation would only utilize one word of a leaf page to map each page (4K/4K = 100%)!

Perhaps a more typical case of the virtual address space requirements for a process:

128K of instructions,
128K of static and dynamic data, and
16K of stack data.

This requires the root page, 1 leaf page for text and data and another leaf page for the stack (3*4K/272K≈4% overhead). The root page might be shared with other root pages and reduce the total overhead.

As the address space grows, the number of table levels needs to grow or the page size needs to increase. Three or four table levels may be needed for even modest growth in the virtual memory range (say 40-48 bits). The best case overhead remains similar. The entry size probably needs to increase to address a large physical address size. In the worst case, the overhead becomes *n* hundreds of percent with *n* being the number of table levels. A fully supported 64-bit address space with 4K pages and 8-byte pointers and entries would require roughly 5 page levels. The time and space implications of large address space systems suggest the consideration of alternative structures and approaches.

2.2 Inverted page tables

Implementations of large address space machines have utilized the inverted page table structure [Lee89] [Chan88] [IBM78]. It is a single table with one entry per physical page. Each entry contains the virtual address currently mapped to a physical page as well as some protection and status bits. Discovering the virtual address given a physical address is trivially determined by indexing the table with the physical page number and examining the entry.

To determine the reverse mapping, namely virtual to physical, a hash structure, the hash anchor table (HAT), is first indexed by some function on the virtual address. The HAT provides a pointer to a linked list of potential IPT entries. A quick linear search comparing the desired virtual address with the IPT entry's virtual address completes the look-up. If no match is found, the virtual address is not mapped and page fault handling is initiated. Figure 2 illustrates this structure.

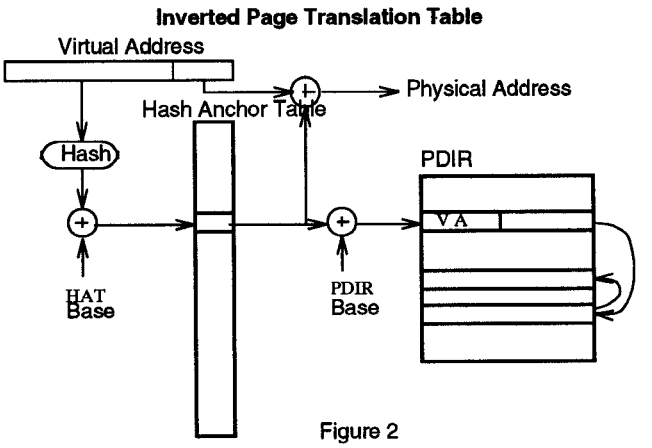


Figure 2

This global table is shared by all the processors. In a sense, the ASID of the forward-mapped table is included in the address itself. Protection in this kind of architecture is either supported by some kind of address isolation or by the use of storage keys.

The biggest difficulty with the IPT is the support for address aliasing. Only one virtual to physical mapping may exist at one time. Whenever aliasing is only used for sharing, most IPT systems accomplish the same function with a global address. To support aliasing for other reasons, the entries must be changed.

Time: TLB miss handling performance is primarily a function of the number of probes to find the translation. The very nature of a hash table suggests one cache miss to reference the HAT pointer. Given a fairly uniform random distribution of virtual to real mappings, each element of the chain is another cache miss. To minimize the average length of the chains, a large HAT is used. Analysis of this type of hash structure allows the designer to trade-off between average hash chain length and number of entries in the HAT. The total cost of the TLB miss involves:

CPU overhead to suspend execution and step
through each part of the algorithm,
Memory reference to the HAT pointer,
Memory reference to the IPT,
Possible memory references for chain elements, and
Possible update to the dirty and reference bits.

It is possible to minimize some of the HAT cache misses for sequential TLB misses, but generally the memory references have high cache miss rates. Later measurements quantify this parameter.

When sharing with global addresses, it is sometimes possible to reduce the total number of TLB misses. Multiple processes can re-use the same TLB entry and avoid misses.

Space: The size of storage for the mappings is a linear function of the amount of physical memory, with an overhead of roughly (size of entry)/(size of page). Independent of the amount of allocated virtual memory, the physical memory overhead for the mappings remains constant. This storage must be contiguous. *Holes* in the physical address space wastes entries in the IPT in order to preserve the index as the physical address. Memory mapped I/O systems can waste significant storage in an IPT if it cannot be efficiently packed.

For example, to map a 32Mbyte physical memory system with 4Kbyte pages, a HAT of 16K entries is used to index a 8K entry IPT. Assume 32-bit physical addresses. 64-bit virtual

addresses can be nicely packed into a 16byte entry for a:
(8x16Kbyte+16K*4byte)/4K ≈ .6% overhead.

2.3 A combined hash table and IPT: The Hashed Page Table

An alternative to the IPT is to combine the hash table and IPT into a single hashed structure, termed - Hashed Page Table (HPT), both time and space improvements to the traditional inverted table are possible. Fewer memory references are required and better utilization of memory is possible. Each entry, HPTE, contains both the virtual address and the physical address. No longer can the physical address be computed from the index. Figure 3 shows the structure of this table.

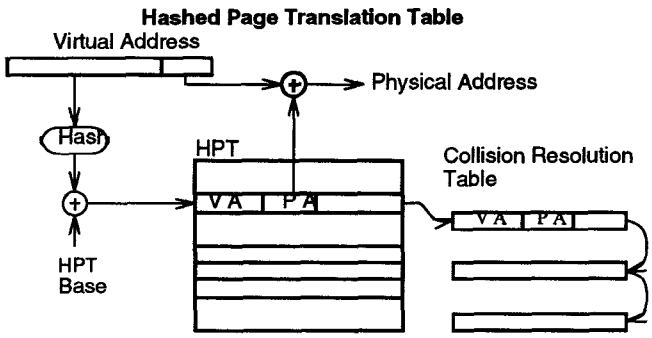


Figure 3

On a TLB miss, some hash of the virtual address is used as an index into the table. The faulting virtual address is compared with the virtual address in the entry. If equal, then the translation is directly loaded. If not equal, then the link is used to chain all of the hash collisions together. Reaching the end of chain indicates a page fault. Collisions can be chained directly into the unused hash entries or chained into an overflow table.

Aliasing is simply supported. Whenever shared global addresses can not be used, the alias is added to the table. This creates multiple dirty and reference bits but does allow different protection attributes. A global address space reduces the need for aliasing and minimizes the number of extra entries.

Time: Servicing a TLB miss requires a reference to the HPT entry and avoids the reference to a separate HAT entry. Eliminating a memory reference is a significant improvement over the IPT structure.

The potential for a chain walk is a function of the size of the HPT. The more entries relative to the number of translations, the lower the likelihood of a chain walk. Choosing a 2 to 1 ratio of entries to physical pages results in average chain lengths of approximately 1.25 entries[Knut73].

Space: Similar to the IPT structure, independent of the virtual memory utilization, there is a fixed overhead that scales with the amount of physical memory. For a table with twice the number of entries as physical pages, the overhead is (16bytes/entry)*2/4Kbytes < 1%. A table with four times the number pages uses < 2% of physical memory.

Address aliasing will reduce the HPT's effectiveness, require a larger size, or require some special handling of certain entries. For example, aliases associated with suspended or swapped processes can migrate to the end of a chain or be deleted and faulted back onto the chain.

The HPT also efficiently handles holes in the physical address space. This has become much more common with graphics adaptors and other I/O devices that take a fixed large amount of address space and use a subset based on the system configuration. This characteristic gives the HPT a significant space advantage over the traditional IPT. For example, a 50 Megabyte un-used segment of physical I/O address space can waste 200Kbytes in unused page directory entries with 4K pages.

Further Discussion: Many variations on the management of a hash table such as the HPT are possible. Collisions can chain to another structure. Secondary hashes could be considered. For example, the MONADS project [Rose85] described a structure similar to the HPT except it was implemented in a separate memory, was managed as a primary TLB, and used internal chaining.

Earlier releases of PA-RISC operating systems, which used the standard IPT structure, had an optional software TLB - swTLB - that is first interrogated using just the low 10 to 12 bits of the virtual page number as an index. An entry in that table was equivalent to an HPT entry for validation purposes, but the swTLB used a secondary hash to resolve collisions into the original IPT. The swTLB proved very effective in early PA-RISC machines for two reasons. The first PA-RISC systems had large direct-mapped hardware TLBs. The software TLB was two-way associative and greatly reduced the cost of thrashing. A slightly later PA-RISC machine had a small two-way associative hardware TLB. The software TLB is very effective since the software TLB had such a high hit rate. This compensated for the lower hit rate in the small hardware TLB, and had faster access because of its simplicity, in CPU overhead, than the IPT table.

By making the HPT very large, it acts as a complete replacement for the older hash table and IPT structure. Alternatively, making the HPT smaller, it acts as a software cache for some other representation of the remainder of the translations. For example, a small HPT can be used in front of a forward-mapped page table.

A final observation: since the HPT entry is nearly identical in form to a TLB entry, it is simple to build a hardware TLB miss handler. It must compute the hash index, accesses the table entry, and faults to software if the entry is not valid or some update to the Such a hardware handler still allows great flexibility for the software. Different page-table policies and organizations are still open to the software since the hardware does not do updates to the table. The designers of a recent PA-RISC CPU chip [Dela92] found the implementation of an HPT hardware miss handler similar to the complexity of the previous generation's 2 level TLB.

By not restricting the size of the HPT or the organization of the overflow table, a VM system has complete flexibility to implement a variety of mechanisms and policies.

3. Data Structures and Implementation

The following sub-sections describe in more detail the format of the measured translation data structures, their requirements, hardware/software interactions, and miss algorithms. Understanding the actual data structures prototyped and simulated along with the operating system environments will aid in the interpretation of the simulation results. This

should help the reader to understand how to adapt the results to alternative data structures and search algorithms.

3.1 Operating systems environments

HP's Unix - HP-UX and the proprietary MPE/iX operating system support text and data sharing through a global address model. Objects are shared between different tasks using the same 64-bit virtual address. It is the responsibility of the operating system to manage the address space to enforce the desired level of protection for each application. Copy-on-write is implemented as copy-on-access. In the MPE/iX environment 64-bit pointers may be directly manipulated by end-user applications. Under MPE/iX the database and the file system are memory mapped and accessed directly through a 64-bit address. Each open file is mapped in its entirety within the 64-bit address space. Both HP-UX and MPE/iX originally used an inverted page table as their primary translation data structure. The trace data used in the simulations was obtained from these two operating system environments.

Most Unix implementations use a more traditional sharing model where each task is given it's own private 32-bit flat address space. If two tasks share code or data, it is accomplished through aliasing multiple virtual addresses to the same physical address. A forward mapped page table is commonly used as the primary translation data structure. These per-task page tables are simple to implement and support arbitrary aliasing. The acronym ASM, for aliased sharing model, is used to represent an operating system that uses aliasing and a forward mapped page table. This model is included in the simulation due to its use in existing implementations of the OSF/Mach based operating system.

3.2 Forward-mapped page table model

A two level page table is selected over a three level table to determine a lower bound on TLB miss times for the ASM model. Deeper forward mapped page tables, which are required for larger than 32-bit address spaces, will only increase the miss overhead. The structure of the simulated forward mapped table and search algorithm is based on TLB miss handler code used in an OSF-1/PA-RISC port. The layout of the ASM forward mapped table simulated is as follows:

Root Table entries

RootEntry

real address of leaf PTE
*
*

Leaf table entries

r	x	t	m	x	rpn[0:19]	00000
r	x	t	m	x	rpn[0:19]	00000
r	x	t	m	x	rpn[0:19]	00000

r= Ref bit, m= Modify bit, rpn= Real page number,
x= Other unrelated bits

Utilizing a per task 32-bit flat address model reduces the amount of data which must be retained in the tables. Per process protection information and high order address bits (upper 32 bits of 64 bit address) can be maintained in a global or control register, and need not be duplicated in each page table entry. The algorithm to handle a user TLB miss is outlined below. Each step may require one or more instructions. Exact cycle counts attributed to each algorithm are provided in the measurement section.

TLB software miss algorithm (user TLB miss): Move the faulting address to general registers. Determine if the reference is to system or user space. Move the User Root table pointer to a general register. Determine the privilege level. Determine if the reference is to a different process's 32-bit space. Calculate the index into the root table. Load the root table entry. Calculate an index into the leaf page. Load Page table entry. Check reference bit. Form protection information. Insert address, rpn, and protection information into the hardware TLB. Finally, return from interrupt.

There is no need for a valid bit in the page table entry. Invalid entries are initialized with an entry which will generate a protection fault if loaded into the TLB and then accessed. The specified algorithm could have been shortened by several cycles if the hardware were capable of delivering system and user TLB miss exceptions on distinct interrupt vectors.

3.3 Inverted page table model

The inverted page table model supports 64-bit global addressing as follows:

HashTableEntry

word0 rpn&link	h	000000	Next PDE Index	00000

PageDirectoryEntry

word0 rpn&link	h	000000	Next PDE Index			00000
word1 tag1	Upper VA					
word2 tag2	Lower VA					000000000000
word3 prot	r	x	t	m	b	rights	0000	key(15)	0

r = Ref bit, m = Modify bit, x = Other software bits

TLB Miss Algorithm:

1. Move 64-bit faulting address and hash table base into general registers. Hash the faulting address and compute an address into the hash table.
2. Load hash table entry (word 0).
3. Check H bit to see if end of chain.
4. Calculate the page directory entry from the page directory base, and hash table pde index loaded in step2.
5. Load virtual address tag1 from page directory word1.
6. Compare faulting address with virtual address tag. If not equal load Next Pde index (word0), goto step 3.
7. Load the virtual address tag2 from page directory entry word2.
8. Compare faulting address with virtual address tag. If not equal load Next Pde index (word0) and goto step 3.
9. Load protection fields (word3), and check reference bit.
10. Insert address, rpn, and protection information into the hardware TLB. Return from interrupt.

The layout of the hash table entry and first word of each page directory are identical. Word 0 of the page directory and hash table encodes both the next link and the physical page number (rpn) for the following entry. While this encoding scheme is more compact and saves memory it also restricted the page directory to a physically contiguous table and allows no aliasing. Each page directory entry resides within one 16-byte cache line. The hash function provides an even distribution of addresses over the hash table.

3.4 Hashed page table model

The format of the HPT is the same regardless of whether it is being used as a native translation table or if it is being used as a cache fronting the ASM forward mapped page tables. The structure supports a full 64-bit global address space and is layed out as follows:

HashedPageTableEntry

word 0 tag1	v	offset[0:14]														space[16:31]															
word 1 prot1	r	x	t	mb	rights				0000	key(15)																					x
word 2 rpn1	0000000	rpn[0:19]																										00000			
word 3 link1	real address of next hpt entry																														

A second 4 word entry was combined with the first for a 32-byte aligned entry when investigating alternate formats.

word 4 tag2	v	offset[0:14]														space[16:31]															
word 5 prot2	r	x	t	mb	rights				0000	key(15)																					x
word 6 rpn2	0000000	rpn[0:19]																										00000			
word 7 link2	real addres of next hpt entry																														

TLB Miss Algorithm:

1. Move 64-bit faulting address and hashed page table base into general registers. Hash the faulting address and compute an address into the HPT.
2. Load HPTE tag word0.
3. Compare faulting address with virtual address tag. If not equal read the next link from word3 and go back to step 2)
4. Load protection fields (word1), and check reference bit. Load the rpn.
5. Insert address, rpn, and protection information into TLB. Return from interrupt.

One or two four word entries are contained in one 32-byte cache line. The preceding data structure was simulated with several variations; two are described in detail:

- 16-byte entries each containing one translation.
- 32-byte entries each containing two independent 16-byte entries which checked in parallel or serially for a match (a 2-way associative HPT).

Each of these was evaluated based on the hardware costs and performance. The best solution for the intermediate hash table will depend, in part, on the hardware organization of the on-chip TLB. A direct mapped on-chip TLB might do better with a two-way associative table.

Optimizations were made to the hash algorithm of both the HPT and inverted page table to further streamline the miss path. The original software hash function was 5 instructions. It was chosen to give an even distribution of addresses in the hash table. It made no assumptions about the address stream. Since the operating system allocates the ASIDs (on PA-RISC an ASID is the upper 32 bits of the 64-bit address and is stored in a space register), rather then just assigning them in a sequential manner, they can be allocated in a pseudo random sequence. This randomization allows a simpler hash function and still approximates a random uniform distribution in the hash table. A single XOR of the upper virtual address bits and the lower virtual page number bits is effective.

The hardware needs to generate a hash table address, so the hash table is aligned to its size (which is a power of two). Hardware can simply OR in the base HPT table address with the hashed index bits to calculate the effective address of the hash bucket.

To reduce the size of an HPT entry representing a 64-bit address, the tag is compressed from 52 bits to 32 bits. This allows a more compact table and requires less overhead to determine if there is a match. To guarantee that the tag is unique in a hash chain the extra bits which are not a part of the page offset must be used in the hash to generate a unique position within the table. This leads to several restrictions on the table. First it can be no smaller then 32 entries (given the 4K page size) and a 48-bit global address space. Each subsequent bit of virtual address space allocated by the operating system requires a doubling of the table size. Even a modest 4K entry table allows the use of 56 bits of virtual address.

3.5 Hardware HPT and the software interface

A final implementation issue is to properly split hardware and software responsibilities to balance the performance, cost, and flexibility. The term *native* HPT is used to describe the scenario where the operating system's translation tables map directly onto the HPT format. Overflow buckets which are searched by software have the same format as the head HPT bucket. When using a hashed page table with HP-UX, hardware searches the head bucket of the operating system's native translation table. On failure, a trap to software allows HP-UX to continue the search. The term *hybrid* HPT describes a scenario where hardware searches an HPT cache and traps to a software managed table if the entry is not found.

Both approaches can be unified in the same hardware handler. The hardware does the same work to search the first bucket of a hash table in either the native or hybrid HPT. The main difference is the action that software takes on a miss. To maximize the benefit an efficient hand-off mechanism is needed which reduces the amount of re-work required when a software trap occurs.

When using the hashed page table as the native tables, the operating system needs to determine where to continue the search. When using a forward-mapped table in conjunction with the hash table, the operating system needs an efficient mechanism to update the hash table once the normal page walk has finished.

Hardware provides the necessary data to the operating system through a control register when it determines it cannot resolve the TLB fault with the entry stored at the front of the HPT.

On a TLB miss, hardware hashes into the HPT and checks for a hit. If the reference bit and modify bit are set to allow the access then the entry is inserted into the TLB. If there is a virtual tag mismatch then hardware deposits word 3 of the HPTE into a control register. Word 3 is not interpreted by hardware and it's value is maintained by software. In the case where the HPT is a part of the native translation table (e.g. HP-UX) word 3 contains the address of the overflow bucket. When the HPT is being used as a cache in conjunction with a foreign table, word 3 will be written by the operating system to point to the entry itself. This will give the software miss handler a handle on where to write the entry in the HPT cache after installing the translation in the hardware TLB.

If hardware detects an invalid virtual tag, a reference or modify bit exception, it traps to software and deposits the address of the head HPT entry into the control register (rather then word 3). In this scenario software needs to inspect the contents of the head bucket. Software attempts to resolve the fault by setting the appropriate bits in the HPT entry and retrying, or trapping to higher level software.

3.6 Software update of table entries

The task of modifying the translation tables in each model is given to software to simplify hardware and give software more flexibility when modifying an entry. It also allows the operating system to keep track of additional information on the types of accesses which are made to a page.

Allowing the operating system to intervene in the first reference and first modification of a translation allows the operating system to break out the standard reference and dirty bits into additional (modified, accessed, and execute) information bits based on the type of access being performed. On systems which have virtually indexed caches and non-coherent I/O systems, this allows important cache flushing optimizations. Not only can this reduce overhead on the single CPU it can reduce communication overhead in an MP system. Without software management of the reference and modification bits this information would have not been possible to collect. The overhead to manage these bits is small given that they were stored in the HPT and are not manipulated in the typical miss path.

4. Measurements

The HPT analysis suggests that it will perform uniformly better than the inverted table. The performance benefits of the HPT when used to cache a different page table structure is not obvious. This section measures and compares the performance of the HPT with the traditional inverted and forward-mapped page table structures. Measurements of the HPT's performance when used as a cache for a forward-mapped page table are also presented.

Data measurements for events such as TLB misses is a difficult task. This work combines 2 common approaches - hardware monitoring and software simulation - to measure meaningful workloads for systems that were not available [Jain91][Ston88].

The selected benchmarks are executed on a specially modified CPU to trace each cycle of execution. It is possible to hold about 2 million instructions worth of continuous execution before the machine must either be stalled or collection suspended while the data is dumped to permanent storage. Stalling the processor creates problems in managing the real time clock and perturbs the measurement. For very simple benchmarks, like the SPEC suite, stalling is acceptable. For more complex benchmarks, like the transaction processing and large multi-user suites, the perturbation of the I/O system would be unacceptable. These traces use statistical sampling. While the system is executing the workload, the hardware tracer captures several traces spaced out in time. The trace includes all executed instructions, instruction addresses, and data reference addresses for that interval: operating system state, user state, interruptions, everything. For this study 20 traces were collected for each workload. The trace were stripped of the TLB misses generated by the measurement system since they correspond to a specific hardware organization. The paper by Jog [Jog90] first describes this environment.

The stripped traces are run through a simulator to mimic the various environments. For example, to simply measure the TLB miss rate, the simulator is configured with the desired size, associativity, and replacement algorithm. Each trace is *executed* by the simulator and the data collected from the TLB simulation. The trace is executed in the sense that the data and instruction memory addresses are applied to a simulation of the target system to capture a variety of relevant measures.

A warm start approximation for the caches is utilized which uses the cache state at the end of one trace as the starting point of the next trace. Measurements of actual systems have validated this approach [Call93].

There were 330 million instructions captured in the traces and over 4.7 billion instructions were simulated.

For this study, the simulator is configured to measure the behavior of TLB miss handling. When a TLB miss occurs, the simulator mimics the memory references that the model requires, generates those addresses, and applies them to the cache and memory models. This two step approach allows the measurement of complex and long running workloads and still retain flexibility in cache, TLB, and page table organization.

Generally speaking, a trace's TLB miss rate is unaffected by the underlying translation structure. But the TLB miss rate is effected by the sharing model. Traditionally, IPT based systems have shared data using common global addresses. Shared instruction and memory segments have the potential to reduce the TLB miss rate by finding a translation from the previous process. This only requires a single TLB entry to exist to map the page for all processes. Forward-mapped page table based systems traditionally share data using address aliasing. Each alias requires an additional TLB entry. When little sharing occurs this is not important but environments with large amounts of instruction or data sharing may encounter different miss rates.

All traces labelled *ASM* are HP-UX traces that have been modified to simulate what would have been the address trace in a per-process address space model. The simulator observes when a context switch occurs, and adjusts the instruction and data address stream to appear to be per-process. This approach generates aliases when the original trace is sharing data or instructions. The most common data sharing is by the instruction segment. Some data sharing occurs in the multi-user benchmarks.

Measurements of the original IPT structure were not modeled using the two steps of tracing and simulation since the benefits for using an HPT over the IPT had already been

demonstrated with prototype software in the lab. At the time this paper was written, resource constraints prevented re-running the traces against just the IPT model. Instead, the IPT data is generated by using an equivalent sized HPT's first bucket cache hit rate as an approximation to the IPT's hash anchor table cache hit rate. This cache cost plus a fixed overhead in instruction cycles is then added to the equivalent sized HPT's total cost.

The following workloads were collected while executing the HP-UX operating system:

- finite - a large finite element application
- doduc - SPEC
- eqntott - SPEC
- espresso - SPEC
- fpppp - SPEC
- gcc - SPEC
- hilo - circuit simulator
- li - SPEC
- matrix - SPEC
- nasker - SPEC
- spice - SPEC
- tomcatv -SPEC
- OLTP1-ux - A large on-line transaction processing relational database application.
- telcom-ux - A telecommunications benchmark.
- OLTP2-ux - A variation on OLTP1-ux.

Additional workloads were collected while executing the MPE/iX operating system:

- Batch-mpe - Batch hierarchical database application.
- OLTP1-mpe - An on-line transaction processing relational database application.
- OLTP2-mpe - A batch manufacturing database application.
- OLTP3-mpe - A variation on OLTP1-mpe.

All the traces - in particular, the matrix and nasker traces - are from older generation compilers and do not represent the latest optimizations. For most programs, this will have little effect on the TLB miss pattern. For matrix, and to a lesser extent nasker, this is a very significant effect. Consider matrix to represent a program that misses the data TLB on every 16 or so instructions.

The last three traces (OLTP1, telcom, and OLTP2) are large multi-user benchmarks and better represent workloads fully utilizing the available memory.

The simulations modeled a 96 entry fully-associative combined TLB. The TLB requires an extra 1 cycle penalty for each page crossing to validate a mapping for the current instruction address. Additionally, block TLB entries map the static portion of the HP-UX operating system and significantly reduce the number of TLB misses. The MPE/iX operating system is paged and does not utilize block TLB entries.

Penalties consistent with current PA-RISC systems are assumed. The hardware portion of TLB miss handling with an HPT takes a basic 9 cycles. For associative table entries, the hardware requires an extra 2 cycles to examine a second entry. The basic software access to the HPT takes 27 cycles. Chain walking takes an extra 9 cycles per chain element. The basic software forward-mapped table access takes 28 cycles and, if necessary requires 8 cycles to update a HPT cache.

All the measurements use the same hash algorithm cost, and equivalent basic cycle costs such that performance differences only reflect the differences in page table structure.

The cache is a 256K direct-mapped data cache and an equal size instruction cache. The average data cache miss penalty is 30 cycles.

In summary, these measurements are derived from actual traces of significant workloads. The results are measured using a simulation of the desired system using instruction traces as stimulation.

4.1 Key to graph labels

The graphs used in the remaining sections are labeled to identify the environment. The label encodes the base operating system type, software or hardware table walking, page table format, HPT size, and associativity. The HPT size ranges from 1/4 the number of physical pages to 4 times the number of physical pages. For example, with 32Megabytes of memory, 16K table entries are 2 times the number of physical 4Kbyte pages.

Name	Type	HW support	Total # of entries
UX-SW-IPT4x-1w	IPT	All SW	32K
UX-SW-HPT4x-1w	HPT	All SW	32K
UX-HW-HPT4x-1w	HPT	HW-HPT SW-overflow	32K
UX-HW-HPT2x-2w	HPT	HW-HPT SW-overflow	32K
ASM-HW-HPT2x-1w	FMPT	HW-HPT SW-FMPT	16K
ASM-HW-HPT1x-1w	FMPT	HW-HPT SW-FMPT	8K
ASM-HW-HPT.5x-1w	FMPT	HW-HPT SW-FMPT	4K
ASM-HW-HPT.25x-1w	FMPT	HW-HPT SW-FMPT	2K
ASM-SW-FMPT	FMPT	All SW No HPT	-
ASM-HW-FMPT	FMPT	All HW No HPT	-
iX-SW-IPT2x-1w	IPT	All SW	32K
iX-HW-HPT2x-1w	HPT	HW-HPT SW overflow	32K
iX-SW-HPT2x-1w	HPT	All SW	32K

Each HP-UX trace was taken on a machine with 32 Megabytes of memory. The MPE/iX workloads were collected on a 64Megabyte machine.

These measurements represent the management of a 48-bit virtual address space. The HP-UX operating system environment allocates the upper 16 bits in a uniform distribution and the lower 32 bits are allocated in the standard instruction, data, and stack segments.

4.2 TLB Overhead Percentage

Graphs 1 and 2 measure the percentage of total cycles per instruction (CPI) attributed to TLB miss activity when utilizing an HPT or IPT data structure. The graphs give insight into the relative importance of the TLB miss component with respect to various workloads. They also demonstrate the impact of an HPT on overall performance. Graph 1 contains HP-UX trace data from both technical and commercial workloads. Graph 2 contains MPE/iX trace data for commercial workloads.

For example, the OLTP1-ux benchmark spends 12% of its time in TLB miss handling. That benchmark runs 3% faster just

due to a software HPT vs. the original IPT structure. Hardware TLB handling gives another 4% improvement. The OLTP3-mpe workload in graph 2 spends 18.5% of its time handling TLB misses under the software IPT model. The software HPT saves 4% and a hardware HPT saves an additional 6%.

Large multi-user programs consistently show the greatest improvement when moving to the HPT. This is to be expected since their more demanding use of address space results in a higher TLB miss rate.

4.3 HPT cache miss rate vs HPT Size

Graph 3 measures the data cache miss rate into the head HPT bucket as a function of the HPT size. Insights into the importance of the cache miss penalty with respect to overall TLB overhead can be obtained by joining data in this graph with graphs 1, 4, and 5. For example, the multi-user workloads show a fairly high TLB overhead in graph 1. From graph 3 these workloads show a cache miss rate of approximately 20% (at 30 cycles per miss). This is a sizeable component of their overall TLB miss penalty given in graph 4.

While the graph does not show the individual cache miss rates for the forward-mapped page table, the following data is presented for comparison. Simulation of the forward mapped page tables under the OLTP1-ux, Telcom, and the OLTP2-ux multi-user workloads showed a 2-3% cache miss rate in the root table, and a 12-15% miss rate in the leaf entry. Under the numeric benchmarks the root table miss rate ranges from a 0-3% while the leaf entries range all the way up to a 25% cache miss rate.

4.4 Original IPT vs HPT

To better understand the TLB miss overheads graph 4 displays the average cycles per TLB miss including cache miss penalties for several configurations all with the same number of total entires. The original software IPT scheme, software 1-way HPT, hardware 1-way HPT, and hardware 2-way HPT are shown. For example, the finite benchmark has a 56 cycle per TLB miss penalty when using a software-only IPT.

The cycles per miss data shows that a software HPT can save a significant number of cycles over the original IPT. The savings can be broken into a static and dynamic component. The HPT saves 6 cycles per miss over the IPT in just basic cycle costs. The remaining difference is attributed to the one less cache line load. The cost of that load is 30 cycles times the cache miss rate into the IPT hash table for the given workload. Those workloads with a modest TLB CPI component, and a high cache miss rate into the translation table, will benefit the most.

In the multi-user benchmarks, the cache penalty cycles accumulated while walking the HPT or IPT amount to 20-35% of the average TLB miss overhead. The use of a hardware HPT miss handler is significantly faster. The basic overhead (not counting cache penalty cycles) is nearly half the software equivalent. The measured dynamic chain lenghts ranged from 1.02 to 1.13 in length.

A 2-way associative HPT performs slightly worse then the 1-way HPT. This was due to the serial compares in hardware (an extra 2 cycles). Had the compares been done in parallel, the higher hit rate in the 2-way table would have made a 2-way table more attractive.

4.5 HPT hybrid vs forward-mapped table

Graph 5 compares various forward-mapped page table (FMPT) results with a native hardware HPT and a small HPT used as a cache for the software managed forward-mapped page tables. The different strategies are compared based on their respective cycles per TLB miss. For example, the small HPT fronting an FMPT is 3 cycles/miss less than a direct hardware FMPT in the *finite* benchmark.

A problem with these HP-UX benchmarks is that they do not use enough of the address space to force the creation of additional levels in the forward-mapped page table (a two level page table is sufficient). A more aggressive use of the 64-bit virtual address space, by the HP-UX operating system, to concurrently map more objects such as is done by MPE/iX would force one or more additional levels to be instantiated in the forward-mapped tables. This would in turn push up the forward-mapped table cycles per miss count.

The results indicate that a small 1-way associative HPT, sitting in front of a forward mapped table, can be effective in reducing the cycles per miss overhead. Under the workloads investigated the combined hybrid strategy gives performance similar to a hardware forward mapped walker, and maintains a simpler hardware structure. This is an important result since it demonstrates that a simple hardware HPT can be designed in a flexible manner which supports/enhances more then one style of page table management.

4.6 TLB miss rate vs sharing Model

With the ASM model, shared memory does not share the same TLB entry. For all the single process benchmarks this makes no difference to the overall TLB miss rate. But with the multi-user benchmarks that share instructions and to a lesser extent data, the TLB miss rate changes significantly.

Graph 6 shows TLB miss rates for the 3 multi-user benchmarks. These small values are magnified by the cost of a miss. The 61% increase in the number of TLB misses for the Telcom-UX benchmark (.35% to .57%) amounts to a proportional change in overall TLB costs from roughly 3.5% to 5%. These numbers do not reflect the use of shared libraries or mapped files that could further increase the sharing of TLB entries across processes.

4.7 Page Table front bucket Hit Rate

Graph 7 measures the hit rate into the head bucket for four different HPT organizations. For example, 82% of the TLB misses are resolved in the front bucket while executing the *doduc* benchmark with the ASM-HW-HPT1x-1w workload.

From the graph, the hit rate into the front bucket is reasonably high. As expected, the larger the table the more likely the first entry holds the desired translation. A 2-way associate HPT achieves a higher hit rate then the 1-way. However as graph 4 demonstrates, the increased hit rate is not enough to offset the extra cycles spent in searching the two entries in series.

The effectiveness of moving an entry to the head bucket on a miss (chain reordering) is apparent when one looks at the hit rate of the much smaller ASM HPT verses the HP-UX HPT (8000 vs 32000 entries). The reason the ASM model can perform almost as well, and occasionally better, is that it is constantly moving the faulting translation into the HPT cache where it is visible to the hardware handler. This suggests that it

might be useful to consider reordering the HP-UX HPT under certain workloads.

Graph 8 shows the sensitivity of the miss rate of the front bucket as a function of the HPT cache size. Generally, the cache is effective, but some workloads - especially the multi-user ones - show the need for large caches to hold the translation working set.

5. Conclusions

This paper demonstrates the effectiveness of a hardware HPT which is flexible enough to be used as the primary translation mechanism for large address space machines or as an efficient cache fronting a different page table design. The HPT is designed to maximize the effectiveness of TLB management by minimizing the overhead in handling TLB misses while still allowing complete operating system VM flexibility. Both hardware and software participate in the HPT trade-offs to provide a cost effective solution.

The analysis and data demonstrate that an HPT will out perform the standard IPT. An HPT maintains the same scalable storage properties as the IPT. This is a significant attribute when managing sparse access patterns.

It is shown that the HPT can be configured to operate like a cache in front of a more traditional forward-mapped table. Not all operating system environments can tolerate the limited aliasing capabilities of a "native" HPT. The data demonstrates that under most work loads the hybrid solution exceeds or equals the performance of the hardware forward-mapped walker.

The measurements reflect the behavior of TLBs in large address space machines in the sense that all the virtual memory of each process is being managed as a single sparsely allocated unit. Since the measured systems contained only 32 or 64 megabytes of physical memory, the measurements are only an approximation of future systems which use a larger virtual address space. The HPT's performance is independent of the physical memory size, the amount of allocated virtual memory, and the sparseness of the virtual memory.

The HPT is being used in HP's latest operating system release on PA-RISC hardware platforms.

6. Acknowledgments

Customizations to the simulator and data collection were performed by Joe Martinka. His comments and insights greatly improved the analysis of this paper. We would also like to thank Eric Delano, Greg Snider, Duncan Weir, John Wilkes, and the anonymous reviewers for their detailed suggestions and criticisms.

7. References

[Apol88] Apollo Computer Inc. *Series 10000 Technical Reference Library Volume 1 - Processors and Instruction Set*. Order No. 011720-A00. Apollo, Chelmsford MA, 1988.

[Chan88] Albert Chang and Mark F. Mergen, 801 Storage: Architecture and Programming, *ACM Transactions on Computer Systems*, Vol 6, No 1, February 1988, pp 28-50.

[Clar85] Doug Clark and Joel Emer. Performance of the VAX-11/780 translation buffer: Simulation and measurement. *ACM Transactions on Computer Systems*, 3(1):31-62, February 1985.

[DEC83] Digital Equipment Corporation. *VAX Architecture Reference Manual.* Doc. EK-VAXAR-RM-002. DEC, Bedford, MA, 1983.

[Dela92] Eric Delano, Will Walker, Jeff Yetter, Mark Forsyth. A High Speed Superscalar PA-RISC Processor. *Spring Compcon '92*, February 24-28, 1992, pp. 116-121.

[Call93] Jim Callister. (in HP's PA-RISC performance group). Personal communication.

[IBM78] IBM. IBM System/38 technical developments. Order no. G580-0237, IBM, Atlanta, GA., 1978.

[Jain91] Jain, R. *The Art of Computer Systems Performance Analysis*, pages 98-101, 404-428. John Wiley & Sons, 1991.

[Jog90] Jog,R., Vitale,P., and Callister, J. Performance Evaluation of a Commercial Cache-Coherent Shared Memory Multiprocessor. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 173-182, May 1990.

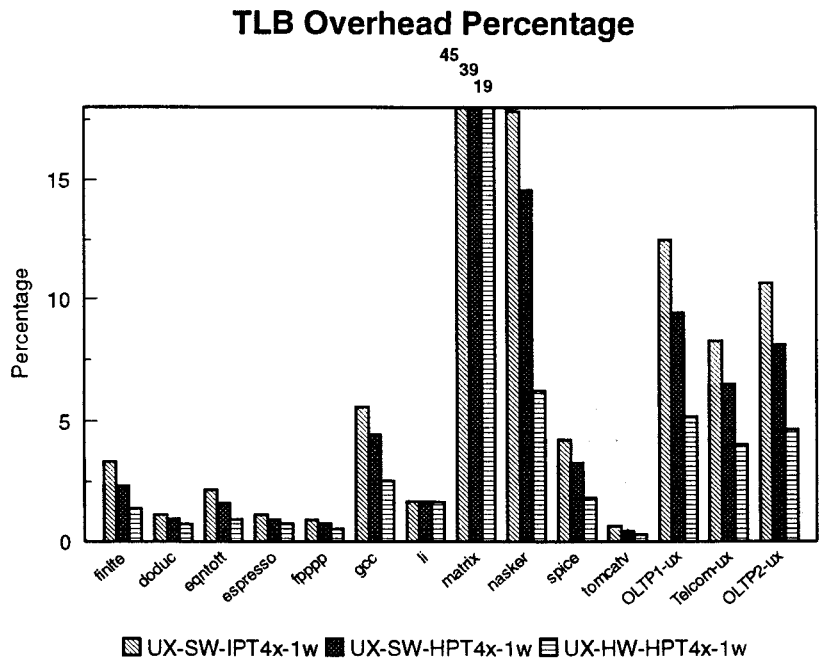
[Knut73] Donald E. Knuth *The Art of Computer Programming - Volume 3: Sorting and Searching*, Addison Wesley, pp 506-549, 1973.

[Lee89] Ruby B. Lee. Precision Architecture. *Computer*, January 1989.

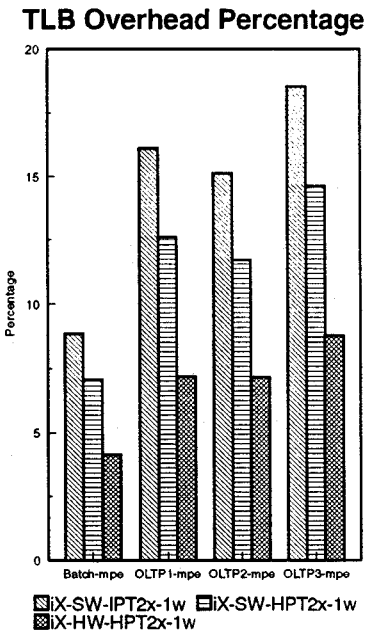
[Rose85] Rosenberg, J. and Abramson, D.A. "MONADS-PC: A Capability Based Workstation to Support Software Engineering", Proc. 18th Hawaii International Conference on System Sciences, 1985, pp. 222-231.

[Ston88] Stone,H., *High-Performance Computer Architecture*, pg. 41-52. Addison-Wesley, 1987.

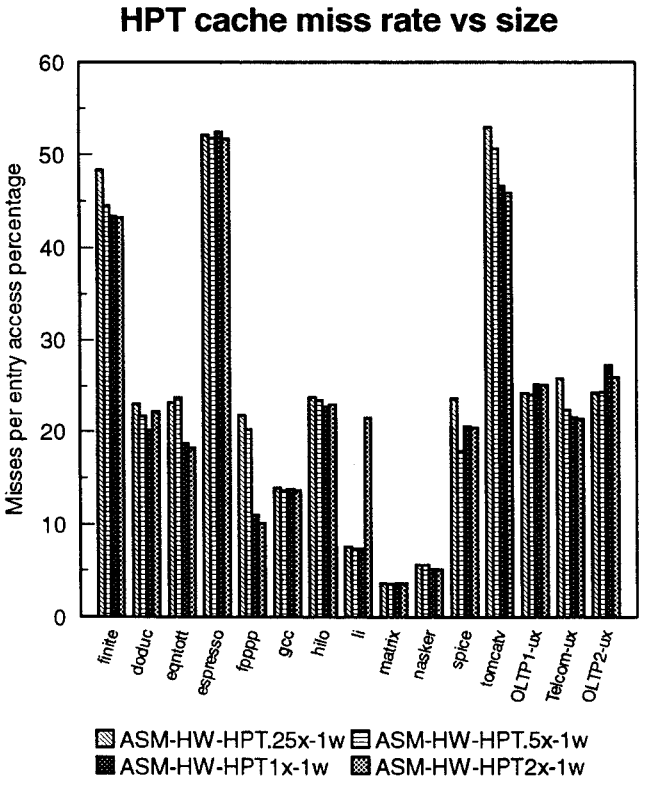
[Tayl90] George Taylor, P Davies, Mike Farmwald. The TLB slice a low-cost high-speed address translation mechanism. In *The 17th Annual International Symposium on Computer Architecture*. May 1990. pp. 355-363.



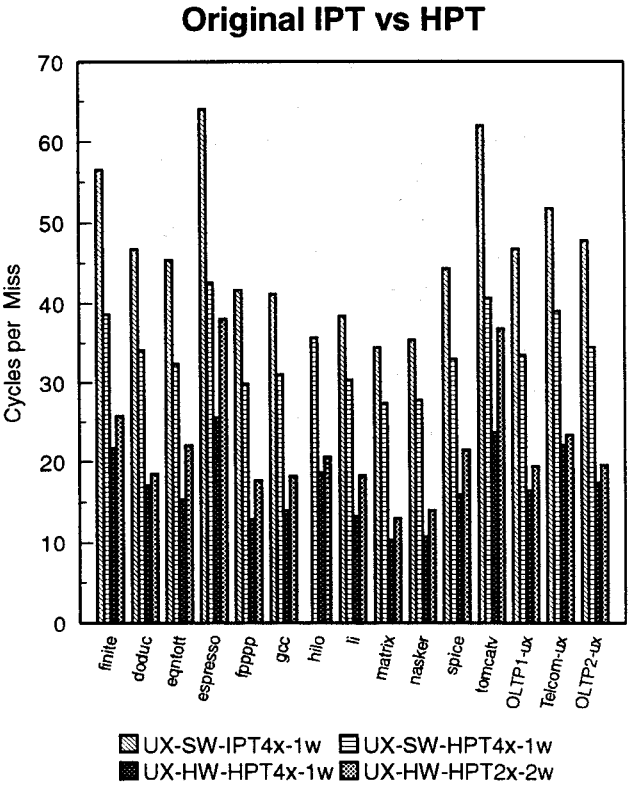
Graph 1



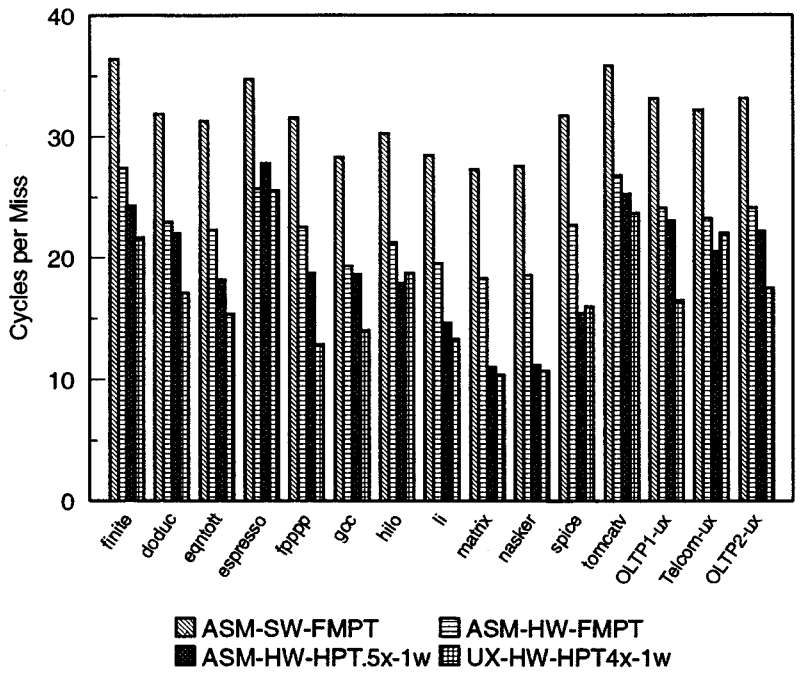
Graph 2



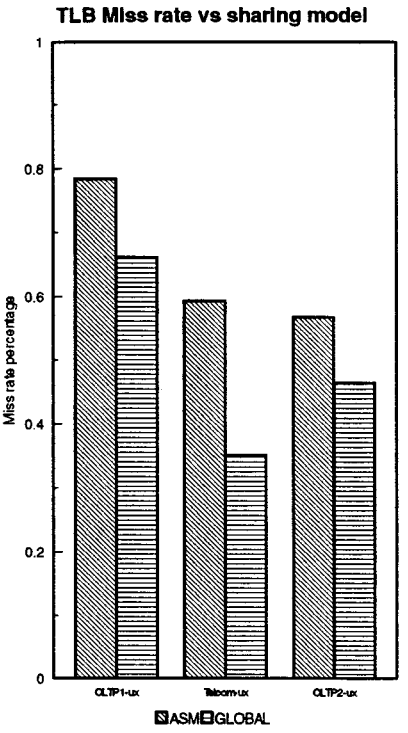
Graph 3



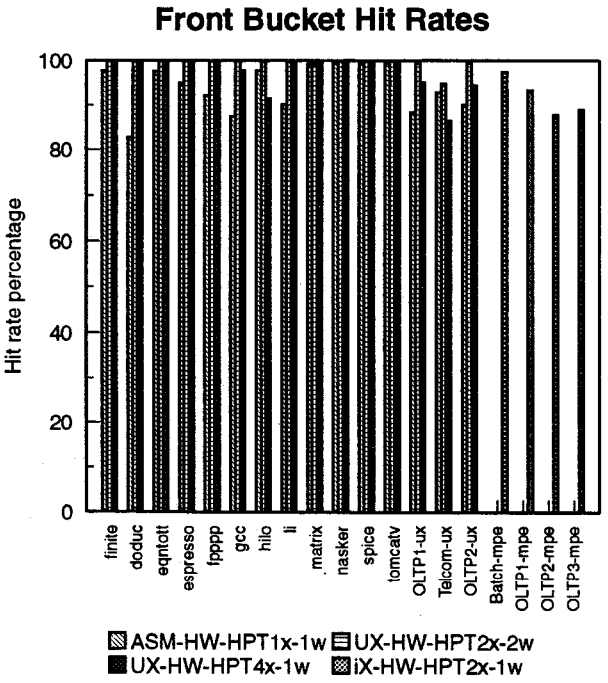
Graph 4



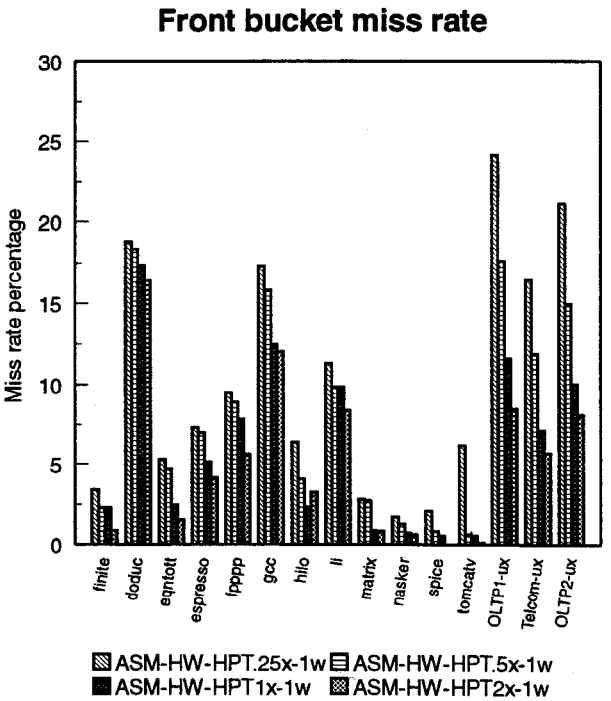
Graph 5



Graph 6



Graph 7



Graph 8