A New Virtual Memory Implementation for L4/MIPS

Cristan Szmajda <cristan@unsw.edu.au>

Undergraduate Thesis

School of Computer Science and Engineering
University of New South Wales
Sydney 2052, Australia

2 November 1999

Thesis Supervisor: Gernot Heiser

Abstract

This thesis adapts a new data structure, the **level and path compressed trie** for use as a page table. Path compression is a technique also applied in the **guarded page table**, used in the present implementation of the L4/MIPS micro-kernel. Level compression is a recent technique which reduces the depth of guarded page tables so that page table look-up in dense address space distributions can potentially be reduced to only one or two array indexing operations.

Like hierarchical page tables, the GPT and LPC trie have natural support for **super-pages** provided by newer architectures. Support for super-pages increases the coverage of the hardware TLB, decreasing the frequency of TLB misses.

The new structure will replace the current virtual memory sub-system of the L4/MIPS micro-kernel. The principle L4/MIPS data structures are also re-designed for efficiency and reduced memory usage. The new implementation is substantially simpler and is designed to be safe in pre-emptable and SMP kernels.

Contents

1
Introduction

This chapter will introduce virtual memory and paging, and explain why modern computer architectures and operating systems require new page table structures.

1.1 Address Translation

In a virtual memory system, every address issued by the CPU is translated into a physical address by the memory management unit (MMU) hardware.

Most virtual memory systems today use paging. In paging virtual memory, the virtual address space is divided into **pages**, which map linearly onto physical units of the same size called **frames**. The function from virtual to physical address can then be represented as a table of translations, one for each page.

Each translation maps a virtual page in a particular address space to a frame of the same size in physical memory.

The page size is always chosen as a power of two. This allows the **page number** to be extracted from an address simply by ignoring the lower-order bits.

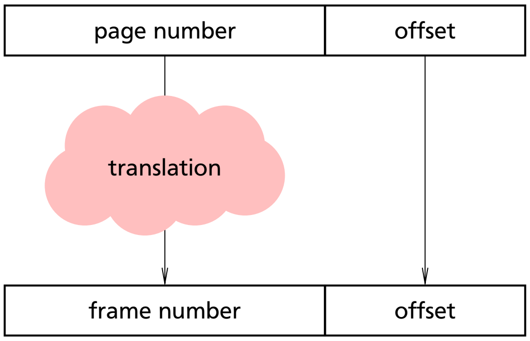

Figure 1.1 Paging Address Translation

The data structure which records all the translations is the **page table**. Given a virtual address and an address space, the corresponding physical address is found by finding the physical address of the matching translation in the page table.

Implementing Address Translation

Translation is a function from virtual to physical addresses, and can therefore be implemented by any of the traditional data structures implementing the **dictionary interface**. However, conventional data structures implementing the dictionary interface are usually inappropriate because virtual memory translation has a unique set of requirements:

- Address spaces have historically been populated in small dense regions spaced far apart. Conventional dictionary data structures are usually optimized for data with a very sparse key distribution such as character strings.
- If different page sizes co-exist then the page table must be prepared to match variable-length keys.
- Dictionary data structures are designed for software implementation. However, the page table look-up has traditionally been performed in hardware. Complex procedural algorithms are

expensive to implement in hardware, though parallel algorithms can be used.

- the page table look-up must be very fast, since it is performed hundreds of thousands of times per second.

Address translation schemes are optimized for programs with localized behaviour, in which a very small number of translations, the **working set** are required at any one time.

The Translation Look-Aside Buffer

Where the page table is too large to fit in MMU registers, it must be stored in main memory. However, if every main memory reference required a translation to be fetched from memory, memory performance would slow down by at least 100%.

The IBM System/370 [8] MMU stored the most recently-used translations in a fully associative cache, which they called the **translation look-aside buffer** (TLB). A TLB dramatically reduces the number of memory accesses for translations in application programs with a small working set. All paging systems today incorporate a TLB.

Since the TLB must be fast, it is usually much smaller than instruction and data caches. Because of the high cost of a miss, the TLB is usually made fully-associative.

On an address space switch, a normal TLB must be flushed, since the translations in one address space are in general not valid in the other address space. A **tagged TLB** marks each translation with an address space identifier (ASID), which is set by the operating system. An address space switch on an architecture with a tagged TLB simply requires changing the address space ID register and the pointer to the new page table.

1.2 Page Table Structures

Inverted Page Tables

The page table on the first virtual memory system, the Atlas computer [6], was an array of virtual addresses indexed by frame number. The look-up was performed by comparing the virtual address with every address in the page table in parallel, as shown in Figure 1.2.

Today a page table indexed by the physical address is called an **inverted page table**, since the table is indexed by the physical address instead of by the virtual address.

On the Atlas, every entry in the inverted page table was compared in parallel by hardware. The inverted page table used in the IBM System/38 [7] was searched with a hash table.

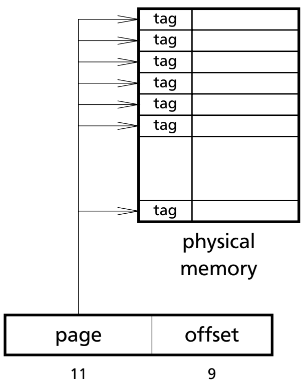

Figure 1.2 Atlas page address registers

The attraction of using an inverted page table is that its size is proportional to the size of physical memory, rather than virtual memory. Many early machines had virtual address spaces smaller than the typical memory size. Nowadays, however, virtual address spaces have leapt from 32 to 64 bits, while physical RAM sizes are growing much more slowly at around 24–28 bits. Furthermore, only one inverted page table is required per system, rather than one page table per address space.

However, it is more difficult to share a frame between address spaces in an inverted page table.

Linear Page Tables

Non-inverted or **forward-mapped** page tables index one or more arrays with the page number to obtain the corresponding physical frame number.

A linear array is the simplest forward-mapped page table, and was used on the PDP-11. The address space of the PDP-11 was 16 bits, divided into eight 8kB pages. The page table of eight translations, indexed by virtual page number, was small enough to fit entirely in MMU registers, even in PDP-11s which provided separate address spaces for code and data.

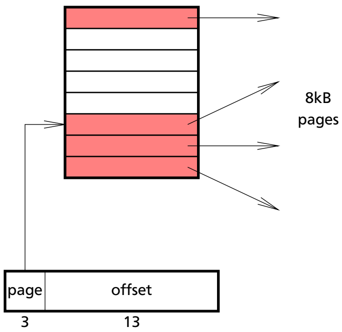

Figure 1.3 PDP-11 MMU registers

Unfortunately, a virtual address space of 32 bits would require a linear page table of over a million entries — far too large for this scheme to work today.

Hierarchical Page Tables

A **hierarchical page table** breaks the page number into fragments which index a hierarchical series of tables in turn.

If an address space was entirely occupied, a hierarchical page table would consume slightly more space than the equivalent linear page table, and would be more costly to look up. Typically, though, large virtual address spaces are only occupied by small regions, so the hierarchical page table saves space overall.

Intel processors from the 80386 [14] use a two-way hierarchical page table for a 4GB linear address space. The 10 most significant bits of the virtual address index the page directory to select a page table, and the next 10 bits index the page table to find the frame number.

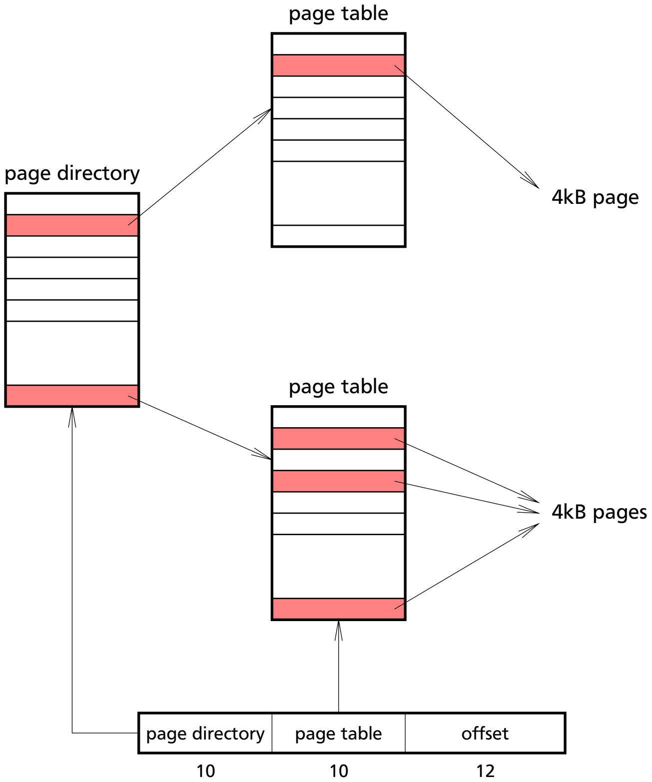

Figure 1.4 Two-way hierarchical page table used on the Intel 80386 [14]

However, for very large addresses the hierarchical page table becomes impractical. For example, a 64-bit address requires a hierarchy five or six levels deep. Attempting to reduce the depth to three would require each table to store 2^{17} entries, occupying 1MB if the entry size was eight bytes.

Virtual Linear Arrays

The **virtual linear array** page table is a linear page table, but contained in *virtual memory*. Faults in the virtual linear array are serviced from a second linear page table, and so on. A maximum of n nested faults can occur can occur before a linear page table in physical memory is reached. For the virtual linear array page tables used on used on the VAX-11/780 [9] and Berkeley SPUR [10], $n = 2$.

Rather than polluting the TLB with entries from the virtual array, the VAX-11/780 included a second TLB for kernel page table translations.

In contrast, the Berkeley SPUR used a combined cache not only for user and kernel TLB entries, but for instructions and data as well. This combined TLB and cache adapted dynamically to the degree of locality in the user code by varying the number of cache entries used for translations. Furthermore, cache coherency algorithms, being pioneered at Berkeley at the time, provided the added bonus of TLB coherency for free.

Effectively, the virtual linear array forms a structure similar to the hierarchical page table, and performs as badly in large and sparse address spaces. However, the virtual linear array is searched from bottom to top, rather than from top to bottom. This reduces the time for typical look-ups by taking advantage of locality behaviour.

On architectures where TLB misses are handled by software routines, which are costly to make re-entrant, the virtual linear array page table is less attractive. Furthermore, TLB entries for the virtual array reduce the number of TLB entries for user pages.

Sorted List Page Tables

An interesting variation on the fully associative page table was used by the CDC STAR 100 [11] series of computers. A fully-associative TLB of 16 registers (AR00–AR15) implemented a least recently used (LRU) replacement policy by moving the matching entry to AR00 on every translation, and shifting all entries above the matching entry down one position.

On a TLB miss, AR00 to AR14 shifted down one position and AR15 (the least recently used entry in the TLB) was evicted and pushed into a region of main memory called the **space table**. A micro-program would then sequentially search the space table, copying each entry down one position until a match or the end of the table was encountered. If a match was found, it was inserted into AR00, and became the most recently used entry.

The TLB was effectively a sorted list in order of recent use, and the space table was an extension of this list in main memory. While a sequential search may seem unattractive, the ability of caches and prefetch buffers to optimize this sequential search is high.

Since the hardware implemented a true LRU replacement policy, TLB conflict misses did not occur in this system. If the working set exceeded TLB capacity by only a small amount, then the sequential search for a TLB capacity miss would usually be very short, as the space table is in sorted order. Furthermore, TLB initialization misses would be minimized, as the top of the space table, which retains the sorted order across context switches, could be loaded into the TLB before each address space switch.

Software Page Tables

Many RISC architectures, beginning with the Stanford MIPS [12], do not define any page table format at all. Instead, they provide only a TLB, which is filled by the operating system with privileged instructions. On a TLB miss, the hardware traps to a software routine which is expected to insert the missing translation into the TLB, and restart the faulting instruction.

Software-loaded TLBs shift the choice of a page table format from the hardware designer to the operating system implementor. While this frees chip area on the CPU to include other features such as a larger cache, or a windowed register file, it can be a significant performance penalty if the TLB refill routine is too slow. TLB refill routines are typically implemented in carefully optimized assembler, since TLB miss rates can exceed 10% of all memory references [?].

Architectures with software-filled TLBs provide some limited hardware support for use by the operating system. For example, a MIPS R4600 instruction inserts an entry into the TLB with a pseudo-random replacement. Implementation of a more sophisticated replacement algorithm is possible but would probably be prohibitively expensive. In particular, implementing a least-recently used (LRU) replacement algorithm in software would not be possible without handling every memory reference in software.

The choice of page table format for a software-filled TLB architecture is therefore large but not limitless. For example, the virtual linear array page table organization is less attractive on the MIPS R4600 because of the high cost of handling nested TLB misses.

1.3 Large Address Spaces

When addresses were 32 bits or smaller, page table structures could be devised using bits of the page number as an array index. For example, the page number of a 4kB page in a 32-bit address space has 20 bits. This page number can be easily split into two 10-bit numbers which index arrays of 1024 elements. If the elements are 4 bytes, then the arrays exactly fit into a 4kB page.

The large space costs of a 64-bit page table would be acceptable if the address space was densely populated. However, 2^{64} bytes is such an astronomically large amount of memory that such address spaces will be under-populated for many years to come.

A page table structure is required that combines the speed and simplicity of array lookup with the ability to deal with page numbers as large as 50 bits.

1.4 Sparsity

Traditional operating systems such as UNIX use an unsophisticated address space model. Each process populates its address space with a text and data segments near the bottom, and a stack segment near the top.

However, modern UNIXes manipulate the address space in a more complicated way. Examples are:

- Shared libraries are mapped into the address space by a library stub procedure the first time it is called. Typically, the whole library binary is mapped into the address space, even if only a few functions from the shared library are required. This results in only a few isolated pages being occupied in a large region of address space.
- Memory-mapped I/O, a technique pioneered in Multics, complicates the structure of the address space, as segments reserved for memory-mapped files must be scattered far apart in a linear address space.
- Shared memory segments also add to the fragmentation of the address space.

For example, Figure 1.5 shows the memory map of an IRIX process on the MIPS R2000/R3000. IRIX shared library support requires the segments to be more spaced out than in the traditional UNIX address space. Furthermore, the shared library segment itself is used sparsely.

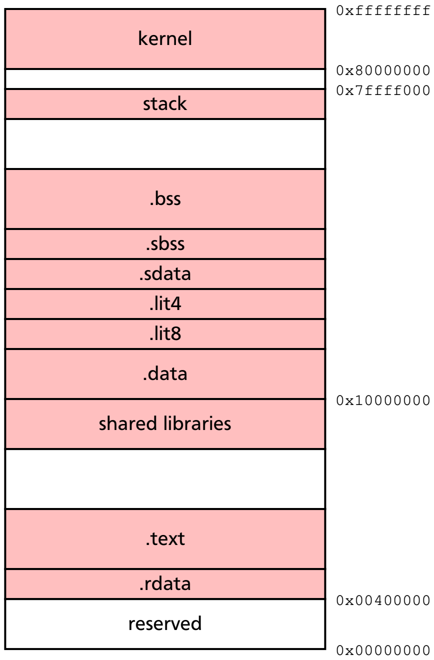

Figure 1.5 Memory map of an IRIX process on the MIPS R2000/R3000 [22]

Single address space operating systems, such as Opal [4] and Mungi [5], push this concept even further. Every resource, such as the file system and network, is accessible via a memory-mapping in a single large address space.

The advantages of this arrangment are that the sharing and persistence of data structures is automatic, context switches can be made cheaper, and shared libraries do not need to use position-independent code. However, these systems use the address space in a very sparse and fragmented fashion.

Hierarchical page tables perform poorly in sparse address distributions, because an isolated page in a large empty region requires whole tables, typically thousands of bytes, to be allocated.

Hashed page tables are better suited to sparse address space usage, since a good hash function will distribute both sparse and dense key distributions randomly. However, hashed page tables do not preserve locality, and have a large fixed creation and deletion cost.

1.5 Super-pages

Common page sizes are 512 bytes (Atlas, VAX-11/780), 4kB (Intel 386), and 8kB (Alpha). The choice of page size is a space-time trade off. Larger pages require fewer translations and increase the coverage of the TLB (discussed later), but smaller pages increase the amount of memory that can be saved through demand-paging.

Newer architectures, such as the Intel Pentium [15] and MIPS R4000 [22], allow more than one page size to co-exist in the same address space. The larger pages, called **super-pages**, have until recently been used by operating systems only for specialized purposes, such as for mapping frame buffers or for kernel's own memory.

Support for super-pages have the potential to reduce the frequency of TLB misses, since the coverage of the TLB is increased. This benefit is most noticeable in architectures where the TLB is refilled by software, and where each TLB entry is tagged with an address space ID, so that TLB entries survive an address space switch.

The addition of super-page support can increase the overall performance of applications by as much as 20% [33].

2
Theory

This chapter will introduce some theory underlying hierarchical page tables, guarded page tables, and the LPC trie.

2.1 Binary Tries

A **binary trie** is a binary tree where branching is determined by the bits of the search key itself, rather than by comparisons between keys. Starting at the most significant bit, a search into a binary trie takes a left branch on a `0` bit, and a right branch on a `1` bit. If the bits of the search key run out and the search has reached a node with a legal value, the search is considered successful. Nodes that do not correspond to valid keys have null or illegal values.

For example, we can encode the English words `a`, `eat`, `hat`, `heat`, `tea`, `that`, and `the` in a binary trie using the encoding `a=00`, `e=01`, `h=10`, `t=11` as follows:

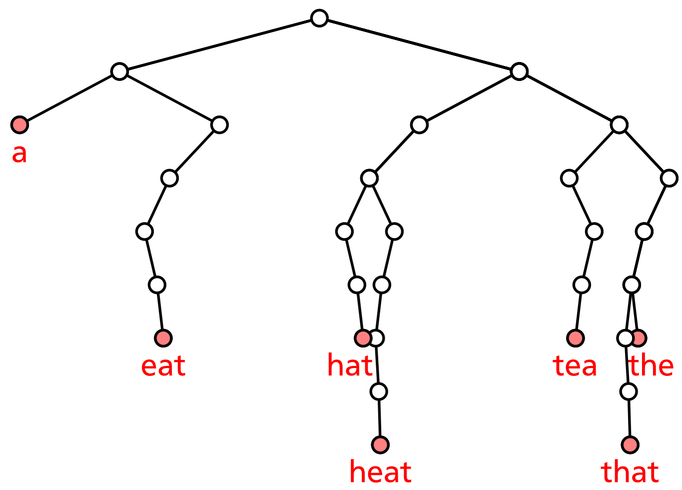

Figure 2.1 A binary trie containing some common English words

A binary trie differs from the familiar **binary search tree** in that it is the bits of the keys themselves, not comparisons between keys, that determine the branch decision.

Because the order of insertion of keys determines the shape of a binary search tree, it can become **unbalanced**. A completely unbalanced binary search tree performs an expensive simulation of a linked list. This worst case occurs frequently in practice — for example, if all the keys are inserted in sorted order.

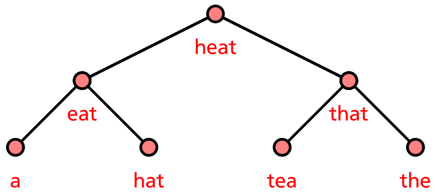

Figure 2.2 A complete binary search tree

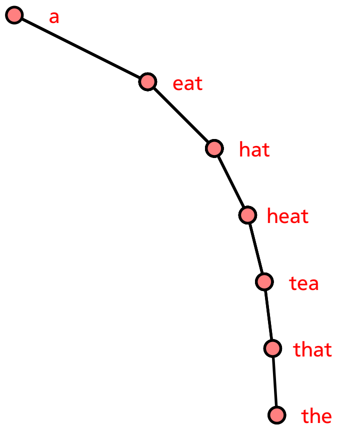

Figure 2.3 An unbalanced binary search tree

Since the position of a key in a binary trie does not depend on comparisons with existing keys, the shape of the structure does not depend on the order of insertion of keys. A binary trie does not become unbalanced like a binary search tree.

Since the depth of a key in a binary trie is just its length in bits, the average depth of a binary trie is the average key length.

A balanced binary search tree will have lower depth than a binary trie. However, each branch in a binary search tree requires a comparison between keys, which is substantially more work than simply testing one bit. Furthermore, in the worst case the depth of a binary search tree containing n keys approaches n. Binary tries do not have this worst case.

Prefix Match

A search into a binary trie succeeds if a valid node is reached *and* the bits of the search key have run out. For example, a search for the word `theta` in the above trie is not successful though a valid node is reached, because the last two letters do not match. String searching requires an **exact match**.

Some applications, however, consider any search which reaches a valid node successful, whether or not all the bits of the key have been used up. For example, in a page table, the keys are page numbers and the searches use virtual addresses. The trailing bits of the virtual address are simply the offset into the page so are not significant in a page table look-up. A match to a key which ignores the trailing bits is a **prefix match**.

If a page table supports only one page size, the size of the prefixes is always the same, and the trailing bits can simply be stripped off, since the size is constant. However, if different size pages are supported, page numbers can be differing numbers of bits, making prefix matches important.

Prefix Keys

In Figure 2.2 no key was the proper prefix of another key. Therefore all the keys (shaded nodes) were in the leaves. If we add some keys which have prefixes already present in the trie, then the prefixes are stored in internal nodes.

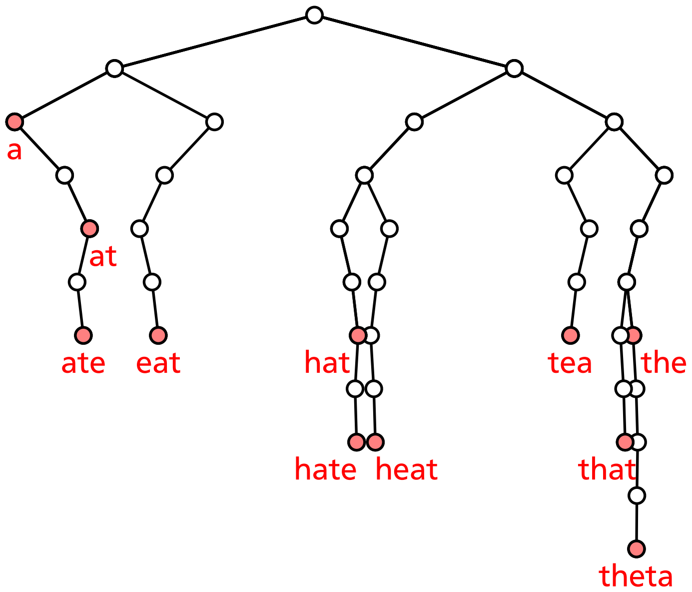

Figure 2.4 A binary trie containing prefix keys

In some applications, prefix keys are implicitly forbidden. For example, a 4kB page in a 32 bit address space has a 20-bit page number. A 4MB page in the same address space would have a 10-bit page number. If the 4MB page number `0010011001` was present in such an address space, any 4kB page with page number starting with `0010011001` would be contained by the larger page. Both keys could not be present in a page table simultaneously.

If no key in the trie is allowed to be a prefix of another, then only the leaf nodes can contain legal values. This restriction allows the storage for the child pointer in the representation of an internal node to be re-used as storage for a key-value pointer in an external node.

A common technique for removing prefixes from string keys is to append a unique end-of-string marker to every key, thus guaranteeing that no key is a prefix of another.

Example: Huffman Tries

Huffman coding [13] relies on the prefix-free property of tries. The Huffman algorithm builds a binary trie of input symbols according to their frequency. The encoding of each symbol is obtained by tracing the path to the symbol's leaf in the trie. Since prefix strings are not allowed in the trie, the compressed binary stream, which consists of encoded symbols concatenated together, can be divided unambiguously into symbols by using the bits to traverse the binary trie from the root and chopping when the search reaches a leaf.

If we construct a Huffman code for the following keys: `ethanal`, `ethane`, `ethanediol`, `ethanoate`, `ethanoic`, `ethanol`, `ethene`, `ether`, `ethereal`, `ethernet`, `ethyl`, `ethylene`, `ethyne`, we get the following coding:

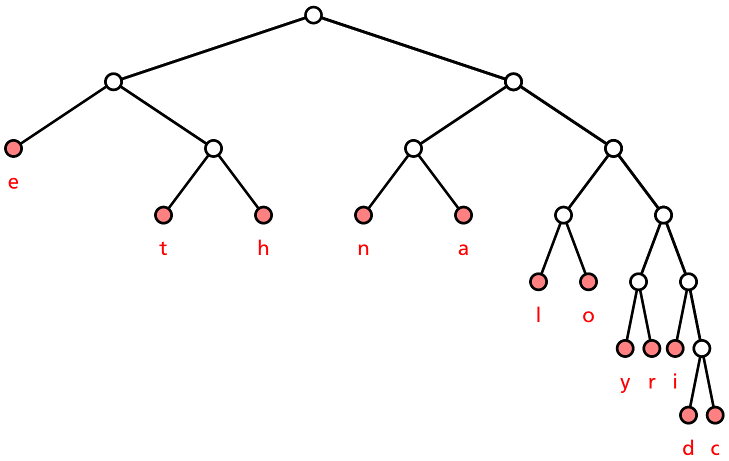

Figure 2.5 Huffman trie for the above keys

The encoding can be read off as `e=00`, `t=010`, `h=011`, `n=100`, `a=101`, `l=1100`, `o=1101`, `y=11100`, `r=11101`, `i=11110`, `d=111110`, `c=111111`.

2.2 *m*-ary Tries

The binary trie can be generalized to tries with bases other than 2. Each key is simply expanded into a string in base m, instead of binary, and each m-ary digit indexes a node with m children.

Example: Quadtrees and Octrees

A **quadtree** is an m-ary trie with $m=4$ used to represent two-dimensional images. The co-ordinates of each pixel in binary are interleaved to form the key. If all the pixels in a sub-trie have the same value, then the entire subtrie can be replaced by a leaf node. The level of detail in the image increases the deeper the quadtree becomes.

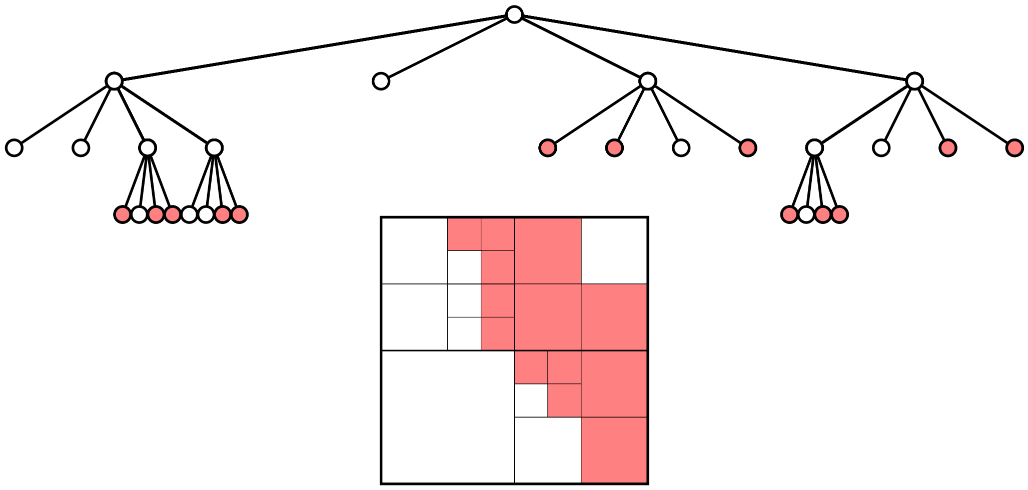

Figure 2.6 A quadtree used to represent a monochrome image.

An **octree** is an m-ary trie with $m=8$ used to represent three-dimensional volumes.

Example: Hierarchical Page Tables

The hierarchical page table (introduced in Section 1.2) is an m-ary trie with m a large power of 2.

For example, the two-level page table used on the Intel 80386 [14] is a trie with base 1024. However, in the Intel Pentium MMU, if a reserved bit in the page directory entry is set, the entry points not to a page table, but to a 2MB or 4MB page [15]. This extension takes advantage of the variable key length allowed by the trie structure. Since a page directory entry points either to a super-page or to a page table of 1024 normal pages, the address space is always prefix-free.

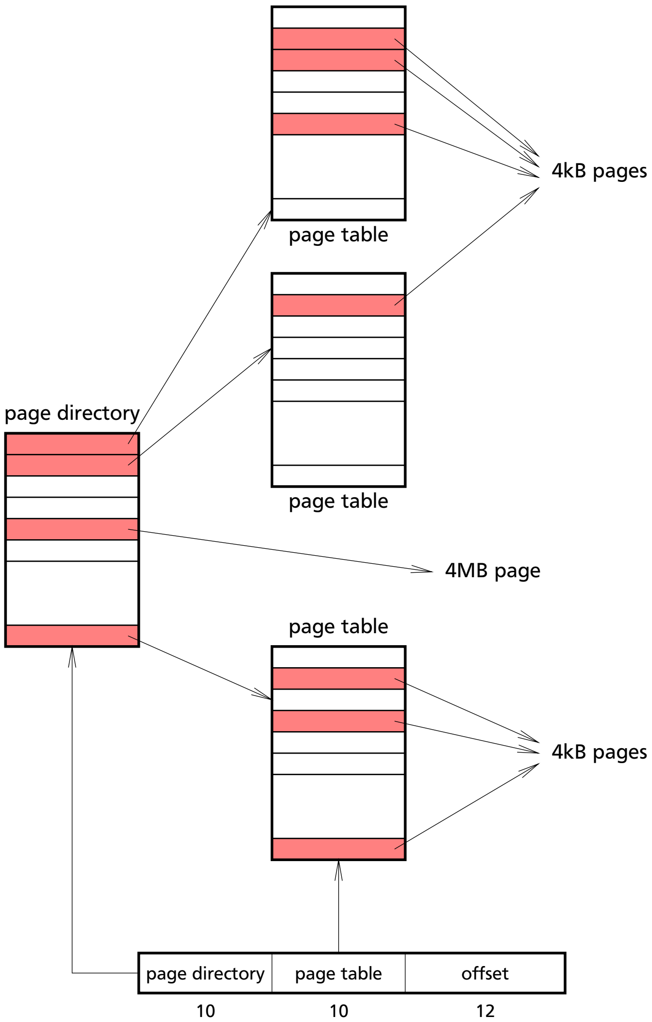

Figure 2.7 The page table structure of the Pentium MMU

The choice of the base m in a hierarchical page table is a trade off between search time and storage space. Large values of m generate a flat hierarchy, but waste space in large nodes, especially if the address space is sparse. Smaller values of m increase the depth of the hierarchy and the look-up time, but are more economical in storage space.

The Motorola MC68851 MMU [34] allows the number of page tables and the size of each table in the hierarchy to be configured by the operating system.

2.3 Path Compression

Tries do not become unbalanced like binary search trees but they are still too deep. The depth of a leaf in a binary trie is equal to its key length in bits.

Long chains of nodes without branches, or **paths**, occur frequently in tries. Where a path occurs in a binary trie, it indicates that all keys below the path have identical bits in the corresponding position. The idea of **path compression** is to convert chains of nodes with one child each into a single node, recording the branches skipped in the path.

Example: Patricia Trees

A **Patricia tree** [16] is a path-compressed binary trie. Patricia trees are used in text searching algorithms [17] and in the the IP routeing table of 4.3BSD UNIX [18].

A Patricia tree containing n keys has n external nodes and $n - 1$ internal nodes. Each internal node has exactly two children, where in the uncompressed trie an internal node may have one or two children. The average depth of a Patricia tree with n nodes is log n, compared with average depth of an uncompressed binary trie, which is the average key length.

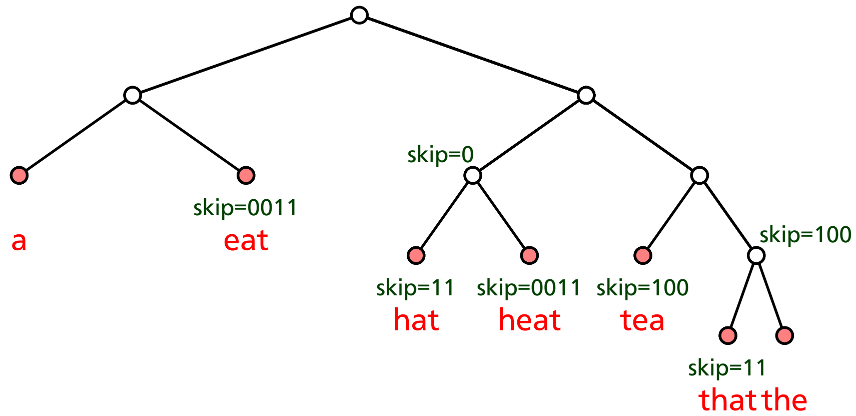

Figure 2.8 Path-compression applied to the above binary trie

Representing the Skipped Bits

The branches skipped can be represented in a variable-length binary string, and compared with the corresponding bits of the key before each branch. However, variable-length bit strings are inconvenient to manipulate in many architectures. Furthermore the keys themselves are often stored in their entirety in a separate table, such as virtual addresses stored in a frame table. It would be more efficient to compare the entire search key with the key in a single comparison, especially if the keys are single words.

So rather than store a variable-length bit string in each internal node, just the *number* of bits skipped need be recorded. The search algorithm then merely shifts the search key by this amount to account for the skipped bits. When a leaf node is reached, the key must be compared with the search key. The comparison will fail if any of the bits skipped were invalid.

Avoiding comparisons at each node decreases the time taken in a successful search at the cost of increasing the time taken to determine that a search has been unsuccessful.

Example: IP Routeing Tables

An Internet router decides where to forward an IP datagram by finding the key in its routeing table that matches the datagram's destination address with the greatest number of bits.

Routeing table keys can be represented in one of two ways:

- the prefix of an IP address in dotted decimal and its length separated by a slash. For example, 192.168.1/24.
- an address extended to 32 bits with zeroes, and a mask of bits to be compared in a routeing decision. For example, 192.168.1.0 with netmask 255.255.255.0.

The key space in a routeing table is *not* prefix-free, because if several routeing keys match a particular address, the most specific match is chosen. The **default route** is a key of length 0, which matches every possible address. However, the default route is only chosen if no other key in the routeing table matches.

One way to arrange the routeing table is to sort the routes with the longest netmask (most specific route) first. A sequential search would then find the most specific match. For example, Figures 2.9 and 2.10 show a network and the routeing table of the host `hildegard`:

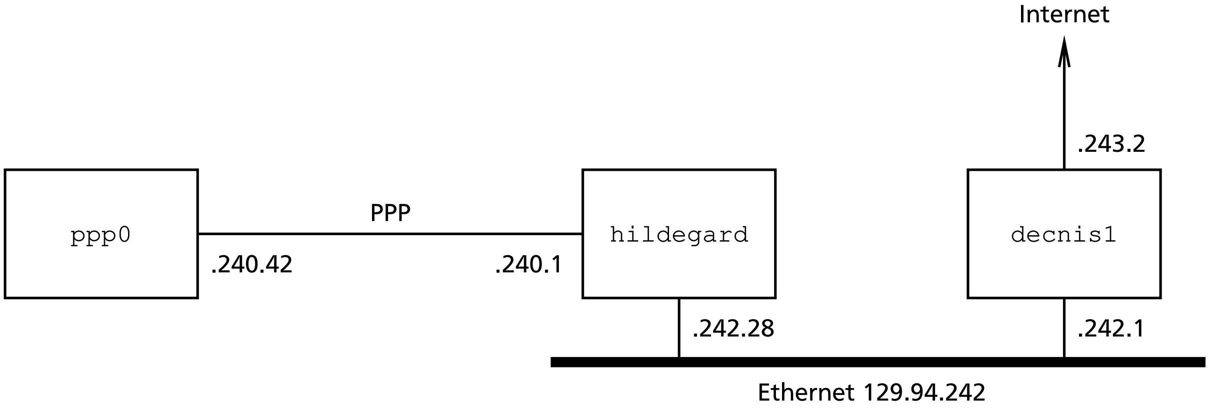

Figure 2.9 Topology of some subnets in the 129.94 network

address prefix	length	destination	gateway
127.0.0.1	32	loopback	
129.94.242.28	32	loopback	
129.94.240.1	32	loopback	
129.94.240.42	32	PPP	
129.94.242.0	24	Ethernet	`decnis1`
0.0.0.0	0	Ethernet	`decnis1`

Figure 2.10 Routeing table of the host `hildegard` in the above network

However, a sequential search through a large table is inefficient. The 4.3BSD implementation of TCP/IP uses a Patricia tree [18].

Figure 2.11 shows the Patricia tree for the above routeing table. The numbers in the nodes indicate which bit to use in the branching decision. The shaded nodes are valid routes. The skipped bits are stored explicitly in the 4.3BSD implementation, and are compared with the bits of the search key in the search. If the comparison fails, the search backtracks to the last encountered route.

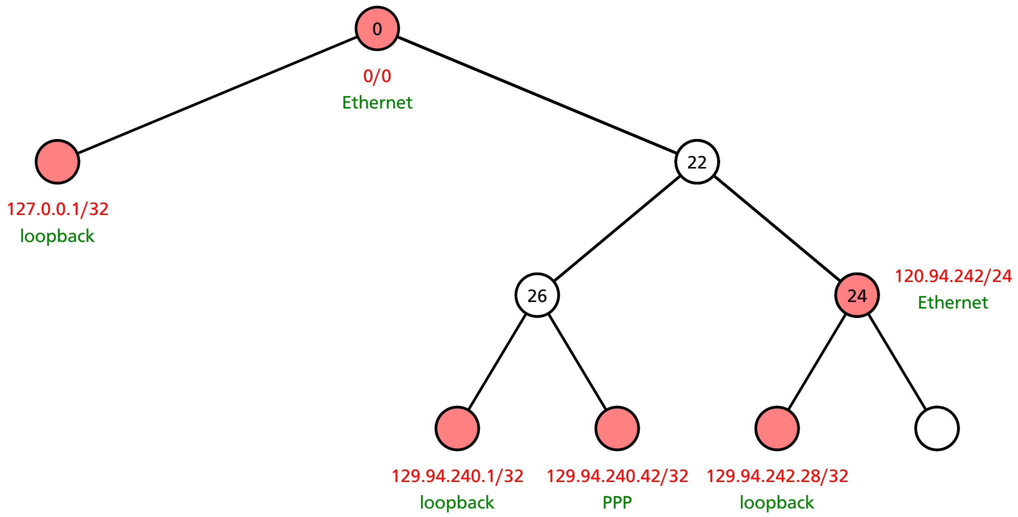

Figure 2.11 Patricia tree routeing table for the host `bsdi` on the above subnet.

For example, consider searching the routeing table for the key 129.94.242.14:

- We reach the root node first, which contains the default route. Bit 0 is on, so the right branch is followed.
- The next node does not contain a key. Bit 22 is on, so the right branch is followed.
- We have reached a matching key of length 24 bits (for the subnet 129.94.242), so the default route can be forgotten. We still proceed further, attempting to find a more specific match. Bit 26 is off, so we follow the left branch.
- This node has a valid key of length 32 bits, but not all 32 bits of the address match. We end the search here. In the 4.3BSD implementation, we backtrack up the tree to the last encountered route, which was the key 129.94.242.

The most specific match encountered was for the key 129.94.242/24. The datagram is sent on the Ethernet interface.

Example: Guarded Page Tables

A **guarded page table** (GPT) [2] is a path-compressed m-ary trie. The bits skipped in a node, termed the **guard**, are stored explicitly in each node, and are compared with the bits of the key in each search iteration. Guarded page tables are used in the L4/MIPS micro-kernel [3].

The guarded page table is a generalization of the Patricia tree to a radix greater than 2. A guarded page table with radix 2 is identical to a Patricia tree, except that the guards are stored explicitly as variable-length bit strings in the nodes.

Figure 2.12 is a visualization of a guarded page table. Unshaded bits are skipped. Each guard is a variable-length binary string which must compare equal with the unshaded bits in the key. The shaded bits are used to index the array of child pointers in each node.

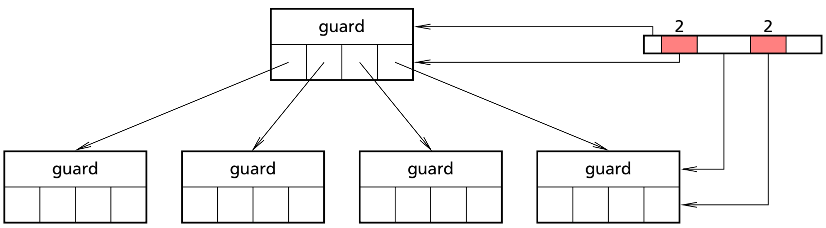

Figure 2.12 Guarded page table with radix 4

As with the hierarchical page table, the choice of the radix m in a guarded page table is a time-space trade off. Larger values of m result in a shallower hierarchy, at the expense of memory usage. However, the radix of a guarded page table is usually chosen smaller than the radix of a hierarchical page table, because bits can be skipped through path compression.

Like the hierarchical page table, the guarded page table supports prefix keys because the structure is order-preserving. This has many benefits:

- Support for super-pages can be added just as in the hierarchical page table. Furthermore, because GPTs typically use a smaller radix the potential for including as many page sizes as those provided by the R4700 MMU is greater.
- Operations on contiguous regions of the address space become easier, as GPTs preserve locality. For example, Schönberg's and Uhlig's Alpha GPT implementation can change the access rights of an entire region of the address space by simply setting and clearing protection bits in the root node of the subtree.

The larger the size of guarded page table nodes, the less likely it is that paths of nodes with only one valid entry occur. Large radixes therefore defeat path compression.

For UNIX-style address distributions, the most likely place for path compression to be effective is on the middle bits of the address, since address spaces are typically populated by widely scattered clumps which differ only in the few most significant bits and several least significant bits of the page number.

Path compression is of no benefit at all for very dense address distributions. For example, the σ_0 address space in L4/MIPS is a radix-16 GPT for all of the contiguous physical addresses available to user applications. On a machine with 64MB of physical memory, such a GPT is seven levels deep.

The radix chosen must be a compromise between saving space for truly sparse address distributions and getting a reasonable depth for dense address distributions.

2.4 Level Compression

Path compression reduces the depth of a trie for a sparse key distribution, by replacing paths of singly-linked nodes with a single node. **Level compression** [1] flattens the regions of the trie with dense key distributions, by replacing complete sub-tries with a single large multi-way node. In level compression, the node size can be a variable power of two, so that larger nodes can be used near a dense region of the key space.

Figure 2.13 shows the effect of level compression on a binary trie. The complete trie on the left can be replaced by a larger radix node on the right. The internal nodes in the left-hand trie may not skip any bits.

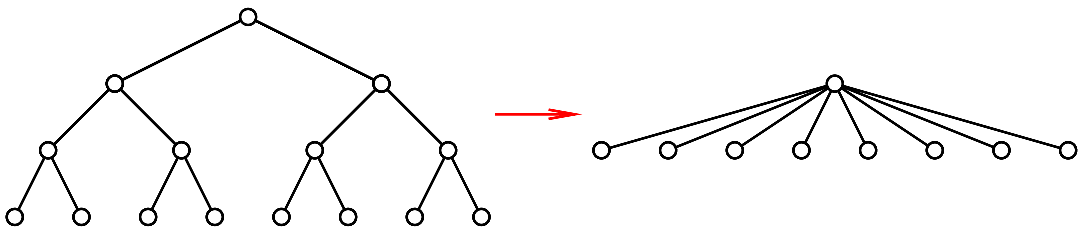

Figure 2.13 Level compression of a complete trie

Level compression reduces the depth of a trie in parts where the key distribution is dense. Path compression reduces the depth of a trie whose key distribution is sparse.

A combined level and path compressed trie is called an **LPC trie** [1]. The diagram below shows the LPC trie formed from the Patricia tree in Section 3.3.

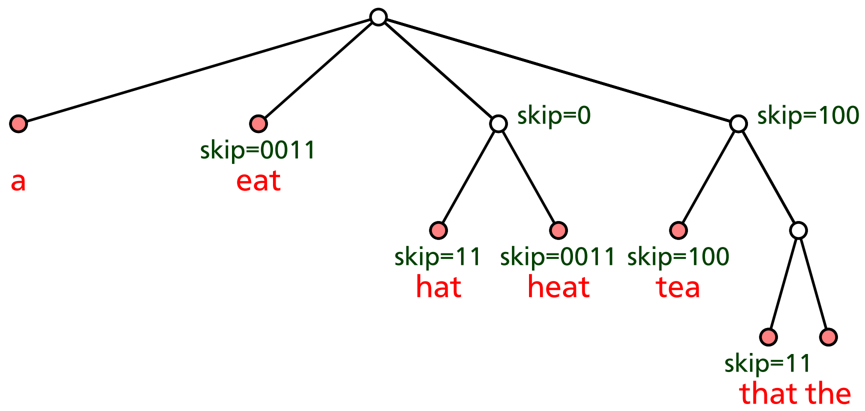

Figure 2.14 A level and path compressed trie

Each node in an LPC trie skips a variable number of bits and branches with a variable number of bits.

For uniform distributions, the expected average depth of a level and path compressed trie is $\Theta(\log^* n)$, where $\log^*$ is the iterated logrithm function: $\log^* 1 = 1$, $\log^* n = 1 + \log^* \lceil \log n \rceil$ for $n > 1$ [19]. This function has extremely slow asymptotic growth.

Storing the Node Size

The size of each node in an LPC trie must be recorded, and the array of child pointers is variable-sized. A straightforward storage layout would be to store the node size at the beginning of the node, followed by a variable-length array.

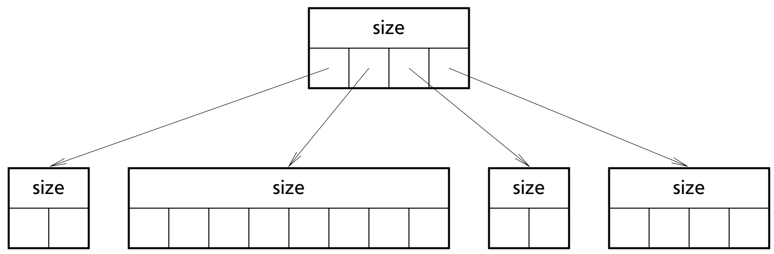

Figure 2.15 Node layout with the `size` member encoding the size of this node

There is, however, an alternative representation. Rather than record the size of the node in the node itself, the size of each node is stored in its parent node, and all the child nodes are allocated consecutively. The difference between these two storage schemes is shown below.

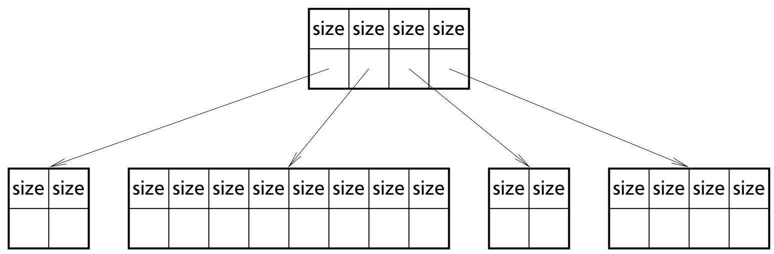

Figure 2.16 Node layout with the `size` member encoding the size of the following node

The difference appears not too great. However in the first layout, the pointer to the next node must be found by using the bits of the key to index an array; in the second layout the pointer is found by adding the bits of the key to the pointer. If the size can be encoded in the same word as the pointer, this involves one less memory reference per search iteration. This is feasible for 64-bit keys since the largest size conceivable can be encoded in 6 bits.

Moreover, the node sizes in the second layout are always powers of 2, which simplifies the storage requirements.

Key Expansion

Like other trie structures, LPC tries can store variable length keys. However, keys that differ in length by only one or two bits defeat level compression. For example, a page table that mixed 4kB and 16kB pages could well use level compression at the root page table, but would fragment down to small size four tables at the leaves.

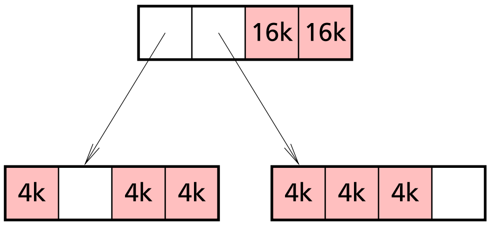

Figure 2.17 Fragmentation caused by keys with similar lengths

One solution is to **expand** odd key lengths into multiple leaves with the same key. For example, if the above page table expanded each 16kB page into four leaves all referencing the same data value, four-entry levels could be joined into more efficient 16-entry levels.

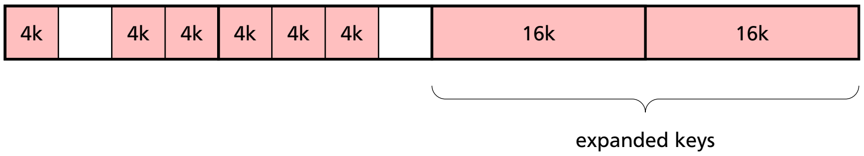

Figure 2.18 Level compression by expanding keys with similar lengths

Key expansion allows level compression to be applied more aggressively, and so saves time at the expense of space.

Key expansion heuristics can be incorporated into the heuristics that decide whether or not to apply level compression. Because key expansion is a dramatic increase in the memory requirements of an LPC trie, a fair rule when counting the number of keys in a node is to consider an expanded key as only one key, even though it occupies more space. For example, 87.5% of the memory in the above table is occupied. However, since there are 8 keys in a 16-entry table, we should consider the node 50% full when deciding whether to apply level compression.

As noted by Talluri [32] replicating the entries is acceptable only for small to medium sized super-pages. Fortunately, very large super-pages can be easily incorporated into an LPC trie withuot undue fragmentation.

Deferred Guard Comparison

Just as the Patricia tree, when searching the LPC trie the skipped bits are not checked until a leaf is reached. This saves space in the internal nodes, but increases the size of leaf nodes. However, since L4/MIPS needs to store the virtual page number of each leaf for other reasons, this costs nothing in extra storage.

The unwieldy storage and checking of variable-length guards in each node of the guarded page table compares favourably with a single full word comparison at the leaf of an LPC trie. This optimizes successful searches at the expense of taking longer to detect a guard mismatch in unsuccessful searches. However, an unsuccessful search in a page table requires costly page fault handling anyway, optimizing the common case of successful page table look-ups is justified.

A drawback of not storing guards in the internal nodes is that it makes sharing whole sub-branches of the trie more difficult, as the shared memory would have to reside at the same address in all address spaces. This is not a problem for implementing a single address space such as Mungi. However, sharing page table entries would require separate protection and flush information for each shared page to be maintained separately, so the idea was excluded from the optimization strategies attempted for this thesis.

Memory Allocation

The difficult part of implementing a tree with variable node size is how to allocate the blocks. With a fixed node size the memory management is much easier.

The memory allocation problem is linked with the problem of when level compression should be applied. If memory is plentiful, level compression should be applied more aggressively. Ideally, the time-space trade off should be applied gradually as the available memory increases and decreases. However, frequent re-structuring of an LPC trie may cost more than the benefit gained through shorter look-up times.

The same trade off is faced by hashing schemes, where the memory cost of a large hash table must be balanced against the cost of re-hashing a dictionary to a larger table.

Even if memory is virtual, expanding node sizes to the point where paging occurs slows down look-up rather than speeding it up.

2.5 Summary

Figure 2.19 is a taxonomy of the data structures in this chapter, with average depths in big-O notation. The quantity k is the number of possible keys (2^{64} for 64-bit keys), and n is the number of keys in the dictionary.

<table>
<tr><th></th><th>no compression</th><th>path compression</th><th>combined level and path compression</th></tr>
<tr><td>base 2</td><td>binary trie
O(log k)</td><td>PATRICIA tree
O(log n)</td><td rowspan="2">LPC-trie
O(log log n)</td></tr>
<tr><td>base m</td><td>quadtree, octree, etc.
O(log k)</td><td>guarded page table
O(log n)</td></tr>
</table>

Figure 2.19 A taxonomy of search structures

3
Algorithms

3.1 Node Representation

The number of skipped bits is not stored explicitly in the internal node. Rather, the `skip` amount is the total number of bits skipped from the root to the branch bits. A left shift by the `skip` amount discards all skipped bits from the search key. Similarly, the `size` quantity is a right shift amount, and is in fact equal to 64 minus the node size in bytes.

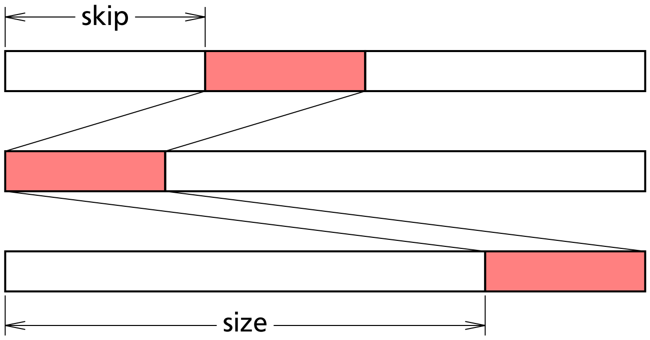

Figure 3.1 Left shift then right shift to extract the index

With 64-bit keys the shift amounts can be encoded in 6 bits each. The pointer to the next node is a physical address, which can be represented in 32 bits on the MIPS R4600.

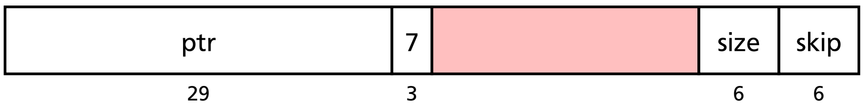

Figure 3.2 Layout of internal LPC trie node

The internal LPC trie node is much simpler than the internal node of a guarded page table, which stores a variable-length guard. Despite its representation in a single 64-bit word, half the size of a GPT node, 20 bits are unused.

The external node is larger than the internal node. The format of the external node is shown below:

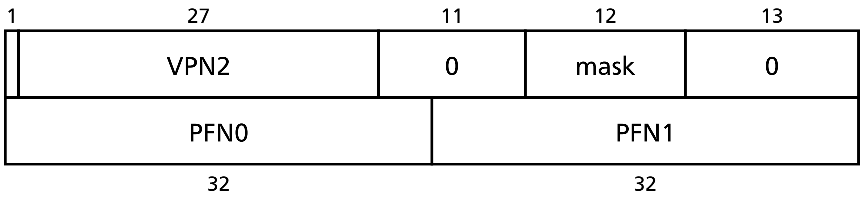

Figure 3.3 Layout of external LPC trie node

Not all of the external node is shown here. The lower half of the external node contains L4-specific information, so will be shown in Chapter 6.

On the MIPS R4600, physical addresses use addresses with a `1` in the sign bit. Therefore, internal nodes are distinguished from external nodes by the sign bit.

Search Algorithm

The search algorithm is in pseudo-code:

> **repeat**
> $node = node.ptr[key << node.skip >> node.size]$
> **until** *node* is a leaf
>
> **if** $key \neq node.vpn2$
>   page fault

Figure 3.4 LPC trie search algorithm in pseudo-code

The search algorithm defers the checking of the guards until after the loop falls through. This version relies on super-page keys being fully expanded, though it is a simple matter to include the code to check the match to a variable-length key.

Compare this algorithm with the pseudo-code algorithm for searching GPTs:

> **repeat** {
> $u = v >> (v_{len} - s)$
> $g = (p + 32u) \rightarrow guard$
> $g_{len} = (p + 32u) \rightarrow guard_{len}$
>   **if** $g == (v >> (v_{len} - s - g_{len}))$ **and** $(2^{g_{len}} - 1)$ {
> $v'_{len} = v_{len} - s - g_{len}$
> $v' = v$ **and** $(2^{v_{len}} - 1)$
> $s' = (p + 32u) \rightarrow size'$
> $p' = (p + 32u) \rightarrow table'$
>   } **else**
>   page fault
> } **until** *p* is a leaf

Figure 3.5 GPT search algorithm in pseudo-code [25]

The GPT search loop, after common sub-expression elimination, contains 17 arithmetic and load operations [25].

Traverse Algorithm

L4 system calls manipulate the page tables in super-page units called **f-pages**, specified by a size and a page number. An algorithm is therefore needed to traverse super-pages in the LPC trie. Recursion must be avoided, since while executing in the micro-kernel the stack size is strictly limited. Rather than using recursive calls, an explicit stack should be used, which saves only necessary variables.

Traversal of the LPC trie requires whole sub-tries to be processed. However, unlike the guarded page table, visiting an internal LPC trie node does not guarantee that the sub-trie does in fact match the super-page being traversed, since bits skipped in the search path may not match.

Furthermore, the super-page might be split over many levels, or might be completely contained by one level. Similarly, if the leaves represent different length keys, the super-page might consist of many leaves, or might be contained in a single large leaf.

The traverse algorithm is adapted from the search algorithm. The search key is the base address of the super-page (the super-page number extended with zeros). Before descending through each internal node, its address is stacked. When a leaf node is reached, it is processed, and the pointer incremented to the end of the level.

Two termination conditions must be observed when advancing over a level. The pointer might increment outside the super-page, or it might fall off the end of the level. Initial approaches to the algorithm required the level size to be stacked along with the current position.

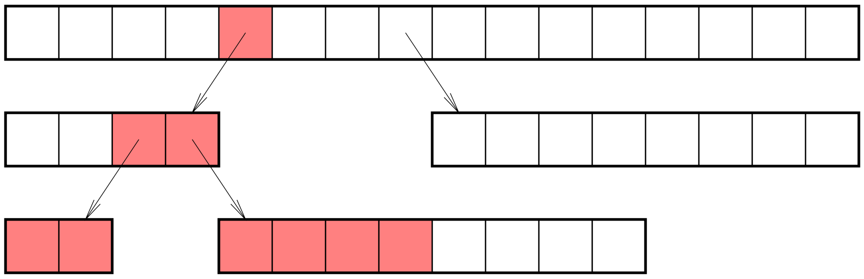

Figure 3.6 Traverse iterates over whole levels, or parts

However, a simpler algorithm was found. Both conditions can be tested by comparing the current position against a boundary pointer which is the smaller of the end of the super-page and the end of the level. How is the boundary pointer calculated? The search algorithm is simultaneously applied to the super-page number extended with *ones*. The fact that ones skipped in the search path do not match the keys is irrelevant, as if we imagine the ones in the key replaced with the matching skipped bits, the resulting pointer is always the highest address that should be processed by this traverse operation.

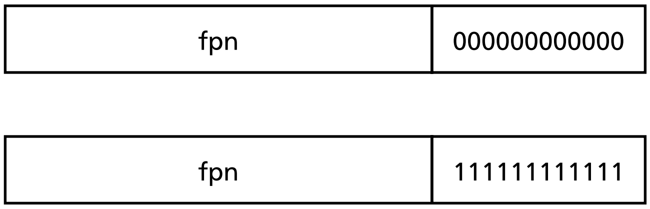

Figure 3.7 Lowest and highest address in super-page

We must therefore be careful to check only bits in the traversed page number with the keys reached in the page table leaves. The check *lop*→*vpn2* >> size ≠ *fpn* in the pseudo-code below does this by using the traverse size, not the page table key size in the comparison.

However, this does not work if a super-page leaf contains the traversed super-page completely. In this case, the super-page size is greater than the traverse size, so the check may incorrectly succeed as if the leaf does not overlap with the traversed region. This condition can, of course, only happen once in the traverse algorithm, so an additional check can be placed outside the main loop.

The pseudo-code for the traverse algorithm for the super-page numbered *fpn* of size *size* is therefore as follows. The pointers *lop* and *hip* are both initialized to the address of the root of the page table.

```
lo = fpn << size
hi = (fpn+1 << size) − 1
lop = hip = root of page table

forever
        repeat {
                push lop, hip on the stack
                lop = lop→ptr[lo << lop→skip >> lop→size]
                hip = lop→ptr[hi << lop→skip >> lop→size]
        } until *lop is a leaf

        repeat {
                if lop→vpn2 >> size ≠ fpn
                        goto out

                process leaf *lop

                lop += sizeof leaf
                while lop == hip {
```

```
                    if stack is empty
                        return
                    pop lop, hip off the stack
                    lop += sizeof node
                }
            } until *lop is a node
out:    if leaf *lop contains fpn completely
            process leaf *lop
```

Figure 3.8 Pseudo-code for traverse operation

Since external and internal nodes are different sizes, the pointer must be incremented by different amounts when advancing across a page table level.

The **forever** loop can be divided into the processing required for internal and external nodes. These cases are separated into two **repeat** ··· **until** loops because a typical traverse will process many consecutive internal nodes followed by many consecutive external nodes. This structure also helps the optimizer save branches in the compiled code.

4
Memory Management

Level compression was initially implemented on static key sets, and did not support insertion and deletion operations. **Dynamic LPC tries** [20], which support the full dictionary interface, must resize their nodes to accommodate insertion and deletion of keys.

The difficult parts of applying level compression dynamically are deciding when to resize the nodes, and how to allocate the required memory. These problems are linked, because level compression can be applied more aggressively if there is more available memory.

4.1 Node Primitives

Dynamic LPC tries must use primitive operations to expand and shrink its nodes. These primitives are **splitting**, **halving**, **joining**, and **doubling**.

Node Splitting and Halving

A node can shrink in two ways. If a branch bit is lost on the left, the node has been **split** in two pieces. If, however, a branch bit is lost on the right, the node has been **halved**.

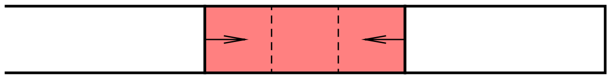

Figure 4.1 The two directions a node can shrink: splitting shrinks from the left; halving shrinks from the right

In node halving, a new array of half the original size is allocated. Every pair of adjacent entries in the original array is considered in turn. If only one entry is valid in each pair, the node it references has its `skip` value incremented by one and the value is copied into the new array. If both entries in the pair are valid, a new size two node with zero `skip` value is created and interposed between the new node and its children.

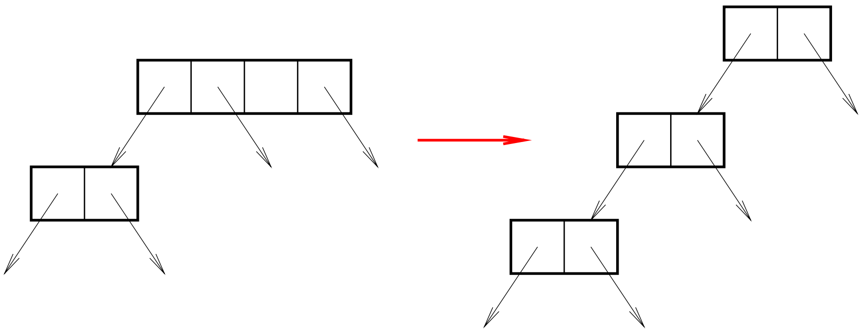

Figure 4.2 Node halving [20]

In contrast, node splitting is much simpler. One two-entry table is inserted above the node, containing pointers to the two pieces. No copying is required, as the node can be split in-place. In the best case, if half of a node is entirely empty, then it can be split without creating any fragments or increasing the trie depth. In general, however, splitting a node adds a two-entry fragment of extra space, and increases the average key depth by one.

Node halving has the potential to create many small two-entry fragments. In the worst case, halving a full node of size n creates n/2 such fragments, increasing overall space utilization by 50%, and the average key depth by one. In general, however, the average key depth is increased by less than one.

Just as in node splitting, in the best case the average key depth is not increased, and space requirements are halved. However, this best case occurs only when every second entry in the array is free, which is a much less likely scenario.

Node Joining and Doubling

Similarly, a node can grow in two ways. If the branch bits absorb a bit to the left, the node has been **joined** in-place to its neighbour. If, however, the branch bits absorb a bit to the right, the node has been **doubled**.

Figure 4.3 The two directions a node can grow: joining grows to the left; doubling grows to the right

In node doubling, a new array of twice the original size is allocated. Each node referenced by the original array has its `skip` value decremented by one. Depending on the value of the skipped bit that was absorbed, the nth node in the original array is copied into the new array in position $2n$ or $2n + 1$. If the `skip` value of a node is zero, it is split, and pointers to the two pieces are copied into position $2n$ and $2n + 1$.

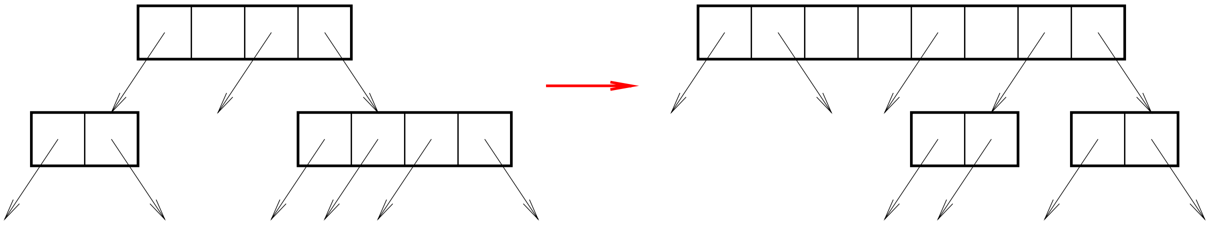

Figure 4.4 Node doubling [20]

Node doubling is much more complex than node joining. Furthermore, the values of the adjacent skipped bits are required. The LPC trie does not store skipped bits in its internal nodes, relying on deferred comparison in the leaves. The only way to find the value of the next skipped bit is to descend the hierarchy and find the first leaf.

Node joining is not as simple as node splitting. If by co-incidence, the two nodes to be joined are allocated at adjacent addresses, *and* the lower address has sufficient alignment, then the nodes can be joined trivially in-place. The next best case would be if the adjacent space to one of the nodes was free. Then one of the nodes would be copied into the free space, and then joined. The worst case would be if a new array was allocated, and both arrays copied. However, this worst case is still better than the best case doubling operation, which always allocates a new array and copies.

4.2 Perfect LPC Tries

A **perfect LPC trie** [20] is an LPC trie with nodes that are completely full. It is possible to maintain a perfect LPC trie by joining only adjacent nodes with empty guards. Removal of a key in the trie results in its node to be split into halves until all nodes are completely full, similarly to the recursive splitting of memory blocks used in the buddy system [21].

A perfect LPC trie contains no invalid pointers. This means that searches will always terminate on a leaf; though the guards must be tested in the leaf, as usual, to determine if the search was successful.

While the criterion for joining and splitting nodes in a perfect LPC trie is clear, the perfect LPC trie applies level compression too conservatively. For example, consider a perfect LPC trie containing the keys 1–31.

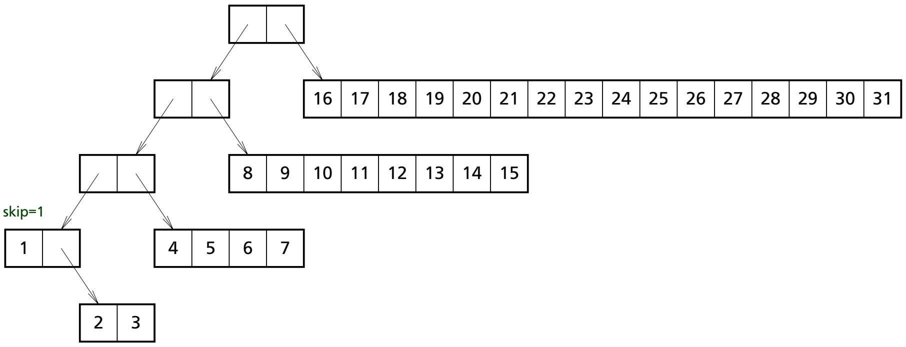

Figure 4.5 Perfect LPC trie for the consecutive keys 1–31

This LPC trie is far from optimal, both in storage and depth. The average key depth is $^{86}/_{31}$ or approximately 2.77.

4.3 Relaxed LPC Tries

The **relaxed LPC trie** [20] allows invalid pointers in the structure. Allowing an invalid pointer for key 0 in the above perfect LPC trie would allow its representation as a single 32-entry node.

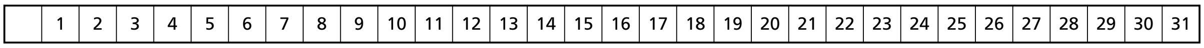

Figure 4.6 Relaxed LPC trie for the consecutive keys 1–31

The relaxed LPC trie uses a threshold for the minimum number of valid pointers in a node before splitting, and a threshold for the minimum number of valid pointers in a combined node before joining.

The higher these thresholds are chosen, the less memory the relaxed LPC trie will occupy, at the expense of the depth. Good thresholds have been determined experimentally at 50% for `low` and 75% for `high` [20].

The drawback of relaxed LPC tries is that an address space being added to page-by-page will undergo joining many times. Joining nodes is comparatively expensive. Consider a 1MB executable being paged in by pure demand paging. Pages will be added to the application's address space one by one, so many node joins will be necessary before the optimal (flat) node will be formed. It would have been easier to allocate a large node to begin with, and split it later if it is not filled. Splitting a node involves no copying, since the half in use remains at the same address, and the unused portion can be returned to the memory manager.

The heuristics employed in the relaxed LPC trie are more suitable for a table with approximately steady-state size, where node doubling and splitting occur relatively infrequently. Fast growth will incur many costly doubling operations as a node is resized from two entries to its desired size.

4.4 A Join-Only Implementation

The perfect and relaxed LPC tries implemented by Andersson and Nilsson [20] were based strictly on node doubling and halving. In order to simplify the doubling operation, they store bit-strings which are essentially guards in the internal nodes, though the guards are not examined by the search loop.

The doubling operation is comparatively expensive. Furthermore, it only allows nodes to grow to the right. This makes sense when adding keys to an LPC trie, because new nodes are typically inserted at the highest bit of difference, and grow downwards. However, if a key is inserted at a higher order bit, a new node will be inserted above to accommodate the new key. It will take many expensive doubling operations for the new node to grow over the old one.

If the join primitive was used instead of doubling, nodes would be completely unable to expand their branch bits to the right. This may not be a serious problem as external nodes never need to grow to the right. However, internal nodes may become fragmented unnecessarily.

Buddy System Allocator

Furthermore, if node growing occurred only through joining, avoiding copying is difficult. Room to grow must be available for nodes to grow. This prompts us to attempt to allocate the nodes as widely-spaced as possible. Initially, an allocator based on the buddy system [21] was devised, using the worst-fit policy to provide the largest separation between nodes.

If a new key was inserted, and no space was available in an existing table, a check was made to see whether the node could be joined with the adjacent (empty) region. If this was so, and the new key would fit in the newly joined node, the memory was allocated and the node enlarged. The check for this case was easy, as it was easy to check whether the highest bit of difference between the new key and existing keys was adjacent to the branch bits.

However, this policy has bad worst case space behaviour. If a new region is built by inserting one key per block, the space required is $O(2^n)$, or exponential.

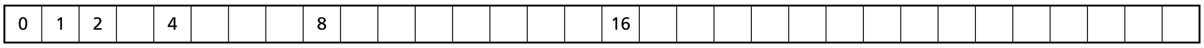

Figure 4.7 Degenerate case

Furthermore, while this allocator separates the nodes, it does not place the nodes with the correct alignment. Joining is not possible in both directions, since in one direction the new node will straddle the natural alignment boundary. The way to determine whether a node can be joined is possible is to compare the lowest bit skipped in the above node with the corresponding bit in the address of the node. If they are equal, the node can be legally doubled.

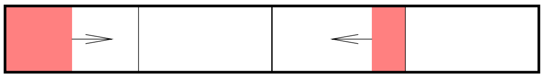

Figure 4.8 Nodes stay naturally aligned only if resized in the right direction

It cannot be assumed that regions will always grow upwards. For example, in the address space of a UNIX process, the data segment typically grows upwards, and the stack segment grows downwards. If an allocator placed the tables for these two segments on 64kB boundaries, each time the stack segment attempted to grow, it would become unaligned.

An allocator was devised that allocates the nodes not just widely-spaced, but aligned so that as many least-significant bits match in the allocation address and the key as possible.

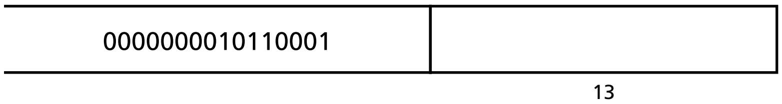

Figure 4.9 Example key prefix

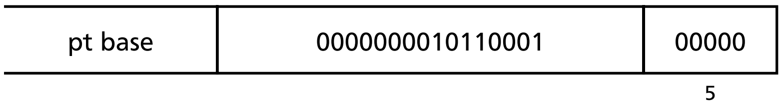

Figure 4.10 Preferred address to allocate table for above prefix

The allocator was told a preferred address to allocate a new table. If this address was in use, an alternative address was chosen which matched in as many low-order bits with the preferred address as possible.

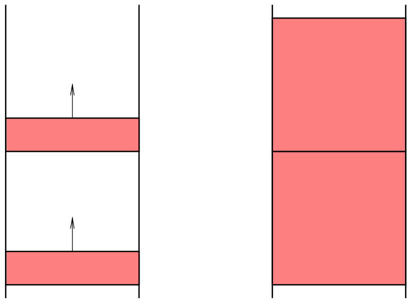

Figure 4.11 A region joined by two regions growing together

However, this search was unoptimized, as a separate search structure to support these preferences was not implemented.

4.5 A Split-Only Implementation

A simpler solution is possible, but it requires that the amount of memory available from the memory manager is known in advance. This is possible in L4/MIPS which currently reserves ⅒ of all physical memory for kernel data structures at boot time.

Rather than only allocating as much memory as is required for small nodes, and then joining them as required, the largest table which can fit in the available memory is allocated, and split when necessary. This requires the memory allocator to know which parts of a node are in use, and which are candidates for splitting.

This policy makes the most efficient use of a fixed allocation of memory. If the L4/MIPS API in the future is changed to allow kernel memory to expand and contract, this policy will need to be rethought.

While the split-only implementation performs well under light loads, it has two major deficiencies:

- Under heavy loads, the level of fragmentation increases to the point where level compression is difficult to apply, even if a compressed trie would save memory overall. While the memory allocator is successful at joining adjacent blocks, the largest available block size rarely rises above a small threshold level when memory is scarce.
- Tables of internal nodes are split just as frequently as tables containing only external nodes. Since level compression in the internal nodes determines the shape and depth of the trie, a better strategy might be to split tables of external nodes first.

An implementation which compacted existing tables to create space for large blocks could solve these deficiencies. However, the benefit from an active compaction strategy probably does not justify the cost.

Bitmap Memory Management

The buddy system uses a binary trie to support the joining of adjacent variable-sized blocks. However, a search into a binary trie requires 10–15 iterations to find small fragments. Small fragments are expected to be used more than enormous blocks. The top-down search of a binary trie costs more as the blocks get smaller, however.

Level compression could be used to compress the binary trie. However, since the free fragments varied in size, level compression would only be effective at the root. Furthermore, the key lengths were of every size from 0 to about 20, so nodes would tend to fragment to size 2.

Furthermore, the buddy system is being used in this case to represent blocks which have been given out but eligible to be revoked. Since the memory is not truly free in general, it cannot be used to store the nodes of the buddy search structure. The search structure would therefore require its own memory allocator.

The solution is to use a bottom-up search, from the leaves of the trie to the root. This is achieved by representing the binary trie as a **complete pre-order traversal** of bits. In the complete pre-order traversal, one bit is recorded in a bitmap for each internal and external node of a complete binary trie. A `1` bit in the bitmap represents used memory. A `0` bit represents memory that is either unreferenced, or referenced but eligible for splitting.

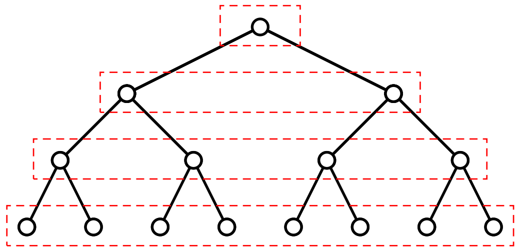

Figure 4.12 Complete binary trie separated into levels

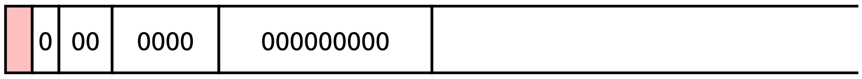

Figure 4.13 Complete pre-order traversal of the binary trie

To determine whether to split a node with a `0` entry in the bitmap, the first word of the block is examined. If it contains an address, it is followed back to the internal node referencing the block. If it contains a zero word, no splitting is necessary.

The smallest allocation unit is 64 bytes, so the bitmap for the lowest level of the binary trie occupies $1/512$ or 0.2% of the memory managed. Each level above the lowest is half the size of the one below it, so that the entire bitmap occupies twice this memory, or 0.39% of the total memory block.

This fixed overhead is small, and makes the memory manager simpler as it requires no memory management of its own.

Navigating around the bitmap representation is quite easy. Shifting the bit index into the bitmap right by one is equivalent to climbing one level up the corresponding trie. Freeing a node involves clearing its bit in the lowest level bitmap, and checking to see whether it can be joined with its neighbour. If so, the index is shifted right by 1, this bit cleared, and the process repeated until a neighbour with a `1` bit is found.

The bitmap must be protected from concurrent access in a pre-emptable or SMP kernel. However, no extra lock variables are required, as updates to the bitmap are atomic bit set and clear operations. Freeing a node must proceed from bottom to top, and allocating a node must set all bits from lowest to highest before using the allocated block.

Updating the bitmap atomically is therefore localized to the region being manipulated. This minimizes memory contention compared with an implementation with centralized locks. Furthermore, the atomic update scheme is safe in a pre-emptable kernel, as no locks are acquired.

Searching the bitmap for an allocation is easy. It is scanned from the beginning, searching for a `0`. This is guaranteed to find the largest available block, as required. To speed up the search as the blocks become fragmented, a global `hint` pointer points to the last position a block was found. The search resumes from this pointer each time. The `hint` pointer need not be concurrency controlled, as it is only a hint designed to speed up access to a block. If it points before or after the truly largest available block, it merely hurts performance.

5
TLB Refill

5.1 Introduction to the MIPS R4600

The MIPS R4600 [22, 23] has thirty-two general purpose 64-bit registers. Two registers, `k0` and `k1`, are reserved for use by the kernel. Arithmetic instructions are either three-register or two-register with a 16-bit immediate. 32-bit constants can be generated in two instructions: a `lui` (load upper immediate) to set the upper 16 bits, followed by an `ori` (bit-wise `or` immediate). 64-bit constants require up to six instructions to generate.

The MIPS has no condition codes. Conditional branches can test whether two registers are equal, or whether a register is negative. Branch and jump instructions do not take effect until after the following instruction (in the **delay slot**) has completed. The R4600 also provides variants of the conditional branch instructions which inhibit the effect of the instruction in the branch delay slot if the branch is *not* taken.

All access to external memory is via load and store instructions. Only one addressing mode, register with constant 16-bit displacement, is available. Signed and unsigned 8-bit, 16-bit, 32-bit, and 64-bit quantities are addressable, but only on natural alignment boundaries. There are two 16kB caches, one for instructions and the other for data, which are two-way set associative with FIFO replacement, and have a 32-byte line size. Stores are queued in a four-word write buffer. An explicit `sync` instruction can flush the write buffer to memory so that memory contents are consistent when read by other processors or devices on the bus.

Because the address calculation and memory access occur at different stages of the MIPS pipeline, the instruction directly following a load may not read the result of the load. However, unlike the jump and branch delay slots, the R4600 will detect this condition and stall the pipeline for one cycle.

The MIPS R4600 includes an on-chip R4700 MMU, with a 48-entry TLB. Each mapping an adjacent pair of 4kB or larger pages. Figure 5.1 shows the format of the R4700 TLB:

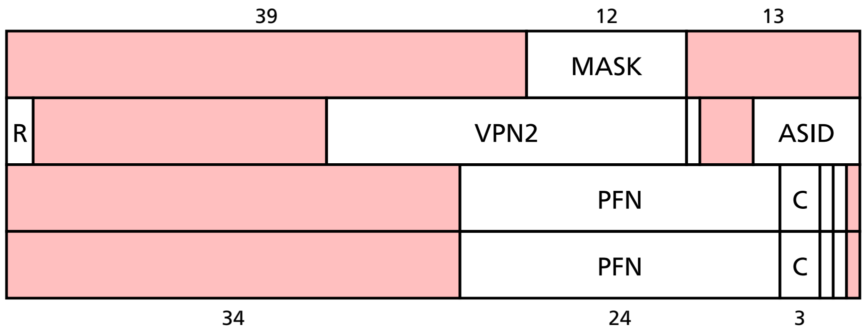

Figure 5.1 R4700 TLB entry format — shaded fields are unused [23]

The page mask specifies the page size in an analagous way to the IP netmask, except that the sense of the bits is inverted. Bits set in the page mask are ignored in the comparison between the VPN and the virtual address to be translated. This inversion is so that a zero page-mask will consider all bits of the page number significant, like a IP netmask of all ones. Not all possible page masks are legal. Only the masks corresponding to 4kB, 16kB, 64kB, 256kB, 1MB, 4MB, and 16MB page sizes are allowed.

The address space must be prefix-free. If multiple TLB entries match, with the same or different number of bits, the operation of the R4700 TLB is undefined.

5.2 LPC Trie Refill

In MIPS assembler code, the algorithm can be coded with 7 instructions in the inner loop, and uses four registers. Figure 5.2 shows the LPC trie TLB refill handler.

```
        lui     k0, KERNEL_BASE
        lw      k1, K_LPC_PTR(k0)
        lwu     k0, K_LPC_SKIP(k0)
        sd      t0, K_TLB_T0_SAVE(k0)
        sd      t1, K_TLB_T1_SAVE(k0)
        dmfc0   t0, C0_XCONTEXT
        dsrl    t1, k0, 6

do:     dsllv   k0, t0, k0
        dsrlv   k0, t1
        or      t1, k1, k0
        lw      k1, LPC_PTR-7(t1)
        lwu     k0, LPC_SKIP-7(t1)
        bltzl   k1, do
        dsrl    t1, k0, 6

        bne     k1, t0, fault           /* test all skipped bits */

        lwu     t0, FLUSH_PFN0-7(t1)
        lwu     t1, FLUSH_PFN1-7(t1)
        dmtc0   k0, C0_PAGEMASK
        dmtc0   t0, C0_ENTRYLO0
        dmtc0   t1, C0_ENTRYLO1
        lui     k0, KERNEL_BASE
        ld      t0, K_TLB_T0_SAVE(k0)
        ld      t1, K_TLB_T1_SAVE(k0)
        tlbwr
        eret
```

Figure 5.2 LPC trie TLB refill routine

Internal and external nodes are distinguished by the sign bit. On the MIPS R4600, a pointer with the sign bit set is sign-extended into a segment of physical RAM. We assume that the root node is always internal, so that no branch is necessary before the first iteration of the inner loop. A truly empty page table must be set up with an invalid external node, however.

Even though the fields of the node may be accessed with one 64-bit load instruction followed by a shift to extract the pointer, two 32-bit loads are used to prevent the MIPS pipeline from stalling.

The MIPS 64-bit shift instructions `dsllv` and `dsrlv` ignore all but the lower 6 bits of the shift amount register. This reduces the number of instructions required to extract the shift amounts, and allows a full 32-bit load to be used, and the value used subsequently after the loop exits.

The `bltzl` instruction inhibits the `t1` register from being overwritten if the branch is not taken. This preserves the pointer to the leaf for use after the loop, and allows the pointer and the `size` shift amount to share a register.

The size is recorded in the internal nodes in bytes, so that the calculated array index is already scaled by 8. The lower 3 bits are aligned by combining the index with the pointer with a bit-wise `or`, and storing ones in the lower 3 bits of the pointer. These bits are masked off by the displacement in the MIPS load instruction, which is in bytes even when it addresses a word. This optimization is specific to MIPS, and saves one cycle in the inner loop. Other architectures, such as Alpha, have load instructions which ignore the alignment bits of the address. On such an architecture, this optimization would not be necessary.

Alignment on the 32-byte boundary for external nodes is ensured by using the R4700 `XContext` register which contains the even virtual page number extended by zeros. The upper 31 bits of this register are unused.

page table base	R	VPN2	0
31	2	27	4

Figure 5.3 MIPS R4700 `XContext` register [23]

Support for the R4700 page-mask costs only one instruction, as the word containing the page mask is available in a register after the loop terminates.

5.3 GPT Refill

The L4/MIPS micro-kernel contains code for many different alternative page table implementations [28]. One of the options in L4/MIPS is for a guarded page table to refill the TLB directly.

Figure 5.4 shows the internal node format for this GPT implementation. Even though the guard is typically a few bits in length, 52 bits are reserved as guards can occasionally skip almost all the bits in the virtual page number.

52	6	6
guard	s0	s1

60	4
ptr	15

Figure 5.4 L4/MIPS GPT internal node [25]

The inner loop of the GPT look-up code is carefully optimized. From a complex loop containing 17 arithmetic and load operations, the inner loop was reduced to eight MIPS instructions [25]. Figure 5.5 is the full direct GPT refill handler.

```
        lui     k0, KERNEL_BASE
        sd      t0, K_TLB_T0_SAVE(k0)
        sd      t1, K_TLB_T1_SAVE(k0)
        ld      k1, K_GPT_POINTER(k0)
        dmfc0   k0, C0_ENTRYHI

        dli     t1, WORDLEN - 4 - GPTROOTSIZE

1:      /* 64 bit GPT parsing loop */
        dsrlv   t1, k0, t1
        or      k1, t1
        ld      t0, -15(k1)
        ld      k1, -7(k1)
        xor     k0, t0
        dsrlv   t1, k0, t0
        beql    t1, zero, 1b
        dsrl    t1, t0, 6

        /* test if match or page fault */
        dsrl    k0, t0, 6
        dsllv   t1, t1, k0
        bne     t1, zero, 2f
        nop
```

```
/* valid match */
lw      t0, (k1)
lw      t1, 4(k1)
dmtc0   t0, C0_ENTRYLO0
dmtc0   t1, C0_ENTRYLO1
lui     k0, KERNEL_BASE
tlbwr
ld      t0, K_TLB_T0_SAVE(k0)
ld      t1, K_TLB_T1_SAVE(k0)
eret
```

Figure 5.5 GPT TLB refill routine [24, 25]

The optimization techniques used were similar to the optimizations detailed above. The pointer member was stored with the lower four bits set to one, and a byte displacement in the load instructions was used to re-align the pointer. Two load instructions are required since the GPT node is 16 bytes, but this also allows the address calculation to overlap in the MIPS pipeline so that the load delay slots are filled. Shift amounts are stored in 6-bit fields with the contents in the upper portion ignored. Finally, a branch-likely instruction preserves the contents of the `t1` register for use after the loop exits.

Despite assuming fixed node size, and including no support for the R4700 page mask, the GPT refill routine is two instructions longer than the LPC trie refill routine, including one instruction in the inner loop. Furthermore, the LPC trie loop is expected to execute fewer iterations in dense address spaces.

5.4 TLB Cache Refill

A technique to improve the average time required for TLB refill is to use a **TLB cache**, also called an **L2 TLB** or a **software TLB** [26, 27]. A TLB cache is a software-maintained cache of translations used by the refill handler to refill the hardware TLB. A true page table must also be maintained, to be looked up if the TLB cache misses.

A direct-mapped 16,384-entry TLB cache forms the critical refill path of L4/MIPS. On a TLB cache miss, the virtual page number is looked up in a normal GPT, and the cache is updated.

The fast path of the TLB cache refill routine from L4/MIPS is included in full below:

```
lui     k0, KERNEL_BASE
sd      t0, K_TLB_T0_SAVE(k0)
lui     k1, TLB2_BASE
dmfc0   k0, C0_ENTRYHI

dsll    t0, k0, 38
dsrl    t0, 47
daddu   k1, t0
ld      t0, (k1)
/* stall */
bne     t0, k0, tlb2_miss
lwu     t0, 8(k1)
lwu     k0, 12(k1)

dmtc0   t0, C0_ENTRYLO0
dmtc0   k0, C0_ENTRYLO1
lui     k0, KERNEL_BASE
tlbwr
ld      t0, K_TLB_T0_SAVE(k0)
eret
```

Figure 5.6 Direct-mapped TLB cache refill routine [24, 25]

The attraction of a TLB cache is that it can be looked up almost as fast as a linear array. However, a true page table of some kind must still be maintained to be looked up when the TLB cache misses. This lengthens the time for TLB refills on a TLB cache miss compared with the TLB refill time for the page table with no TLB cache.

The TLB cache must also be kept consistent whenever translations are deleted from the page table. Furthermore, a TLB cache increases the overall memory required by the virtual memory sub-system.

Frequency of TLB Cache Hits

A TLB cache improves the average performance of the TLB refill path if the number of cycles saved on cache hits exceeds the number of cycles spent handling cache misses. The TLB cache technique is only justified if the hit rate is high and the miss penalty relatively low.

If the TLB cache is made sufficiently large, capacity misses are infrequent. However, since the TLB cache is typically direct-mapped, conflict misses dominate. A conflict occurs in the TLB cache if the same virtual address is used in more than one address space. Conflict even occurs in L4 if the page is truly shared between the address spaces, since L4 treats the aliases as distinct mappings which happen to share the same physical address. Contention will also occur if the difference between two virtual addresses is a multiple of 128MB, which is a realistic possibility in single address space systems.

Conflicts between address spaces could be resolved by hashing the TLB cache tags with the address space ID. Another way to reduce conflicts is by increasing the TLB cache associativity. However, in software increasing the associativity is expensive, as the tags are checked sequentially. Increasing associativity complicates the code further, as a pseudo-random replacement policy must be implemented. It also lengthens the time taken in maintaining TLB coherency. Figure 5.7 shows the costs of implementing some TLB cache algorithms.

associativity	**cache tag**	
	virtual address	**virtual address and ASID**
direct-mapped	23	26
2-way associative	26	28
4-way associative	28·5	30·5
8-way associative	32·75	34·75

Figure 5.7 Average best-case costs of some L4/MIPS TLB cache refill routines [25]

As the above table demonstrates, the more sophisticated the TLB cache algorithm, the smaller the benefit of the TLB cache technique over a full page table look-up. Recall that the best-case LPC trie look-up costs 30 cycles, plus 7 cycles for each extra internal node traversed.

Furthermore, the above analysis neglects the effects of data and instruction cache misses. Unfortunately, the code measuring the cache penalty in micro-benchmarks can itself be the cause of cache misses. Also, the benefit of a cache-friendly algorithm is felt not just in the TLB refill routine but in the surrounding user code. Hence the only reliable way to measure such effects is with macro-benchmarks, which are presently unavailable.

However, shorter code which touches fewer memory locations will in general have a smaller cache footprint. This argument speaks in favour of the simpler TLB cache algorithms.

Comprehensive measurement of TLB cache hit rates under realistic workloads would provide final justification of the TLB cache technique when compared with carefully optimized, state-of-the-art page tables.

TLB Cache and Super-Pages

The TLB cache in L4/MIPS implicitly assumes translations map two 4kB pages, and so is incompatible with the addition of super-page support. Three choices are available:

- Store super-pages as many keys in the software TLB. This will have a neutral effect on TLB cache capacity. However, setting the R4700 page mask support lengthens the TLB refill handler by two instructions, and increases the cost of maintaining TLB cache consistency when a super-page is flushed.
- Do not insert super-pages into the software TLB. A TLB refill for a super-page would always miss in the software TLB, requiring the slow path through the page table look-up. This will allow super-pages to provide the benefit of increased TLB coverage, without lengthening the refill path for applications which take no advantage of super-pages.
- Abandon the TLB cache altogether and use the LPC trie alone.

TLB Cache and SMP

For SMP kernels, TLB coherency is a problem. After a translation is deleted, each CPU which has ever scheduled a thread from that address space must be interrupted to execute a routine which flushes stale entries in its TLB.

The following choices are available for the TLB cache:

- Use one global TLB cache. This option will probably lead to unacceptable memory contention. Furthermore, translations must be updated in the TLB cache in atomic operations, thus complicating code in the TLB refill handler.
- Use one TLB cache per CPU. There must then be one copy of the code for the TLB refill handler per CPU, as there will be multiple base addresses of the TLB cache. This will also increase the cost of synchronizing TLB caches, as multiple TLB caches must be flushed.

While the TLB refill handler needs no modification in a pre-emptable kernel, since it can safely run with interrupts disabled, TLB cache flushes must be made atomic.

5.5 Conclusion

While the best case for TLB cache refill (23 cycles) is better than the best case for LPC trie refill (30 cycles), the high cost of TLB cache misses, the lack of super-page support, and the cost of TLB cache coherency may limit the performance of the TLB cache technique. Modifications to reduce these costs narrow the advantage of the TLB cache even further.

The problem is that TLB cache hit rates have not been measured under realistic workloads. Measurements of the cost of maintaining TLB cache consistency should also be made, especially if super-pages are to be included in the TLB cache.

Careful optimization of the assembler refill routine has a substantial effect on the final refill cost. The L4/MIPS GPT implementaton has been optimized to 50 cycles for a three level look-up, with an inner loop of 8 instructions. Compare this with the L4/Alpha GPT implementation which costs 83 cycles for a three level look-up on the 21064 processor [29]. The inner loop has 12 instructions, costing 13 cycles. Yet this implementation is optimized by 25% compared with an earlier unoptimized version.

Level compression on dense address spaces in the LPC trie promises to reduce the hierarchy to a depth of one or two. Together with a carefully optimized inner loop of 7 instructions, a 37 cycle cost for a two level look-up is fast enough to be used without a TLB cache. Together with the potential reduction in TLB miss rates due to super-page support, the LPC trie is a viable alternative to traditional page tables.

6
The L4 Virtual Memory Model

L4/MIPS provides three primitives for the manipulation of address spaces. The unit of manipulation in L4 is the **f-page**, an 2^n byte super-page with 2^n byte alignment. The operations supported by L4 are as follows:

- **Map**, or copy of an f-page from one address space to another. The sender task specifies the source address and the receiver task specifies the destination address, which may be different. A successful map operation establishes shared memory between the two address spaces.
- **Grant**, or moving an f-page from one address space to another. This is similar to the map operation except that the source f-page also disappears from the sender's address space.
- **Flush**, or removal of an f-page from all address spaces to which it has been mapped. There is no way in L4 to flush a page from one particular address space and not others.

The size of an f-page can be any power of two from the minimum hardware page size to the size of the entire address space.

The three L4 primitives can be used to construct arbitrary address space hierarchies. For example, Figure 6.1 shows some address spaces constructed using the **map** operation.

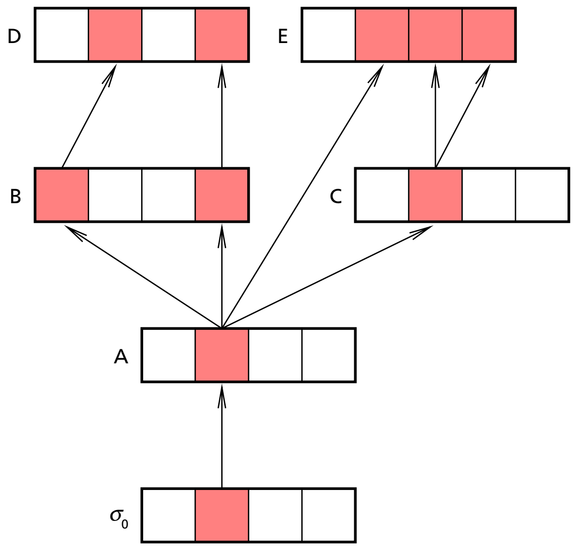

Figure 6.1 Address space composition in L4 [28]

As shown in Figure 6.1, L4 even allows pages to be mapped multiple times in one address space, effectively creating aliases, even though no useful purpose is known to exist for this arrangement.

The σ_0 address space is initialized to contain an identical mapping from virtual to physical addresses. Initial applications running on L4 are mapped pages by σ_0 when they are first touched. This is currently done in 4kB units. Allowing address spaces other than σ_0 to make use of super-pages will require an API change to L4/MIPS.

F-pages encourage L4/MIPS applications to make use of super-page support. However, if L4/MIPS applications use minimum page sizes exclusively, no super-page support occurs, even if adjacent frames are eligible to be joined into a super-page. This is because joining would require additional checks to be added to every map operation, thus increasing the cost for virtual memory operations which do not involve super-pages.

6.1 Implementing Flush

In order to implement the flush operation, the L4 kernel maintains a database recording the family tree of each frame. Currently in L4/MIPS, this is implemented by allocating one 24-byte node per page linked in a tree.

```
typedef struct _t_n {
  udw_t vaddr;
  w_t task;
  w_t child;
  w_t sister;
  w_t parent;
} mt_node_t;
```

Figure 6.2 Flush node declaration in the present L4/MIPS [24]

Figure 6.3 shows an example L4/MIPS flush tree structure. The thick vertical arrows show `child` pointers, the thin horizontal lines are `sibling` pointers, and the parent pointers are not shown.

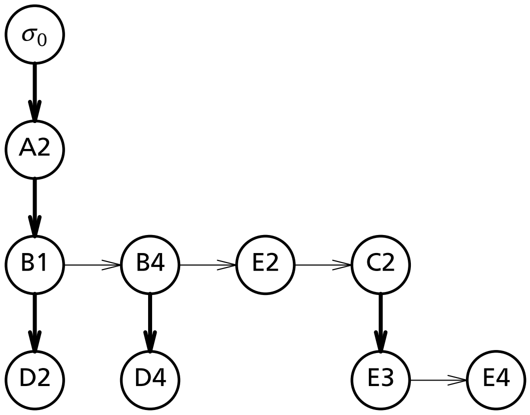

Figure 6.3 An L4/MIPS flush tree [28]

A flush operation prunes a branch from this flush tree. The implementation descends the subtree via the `child` pointer if any, then iterates across all of that child's `sisters`. Instead of returning via the `parent` pointer, the current L4/MIPS implementation pushes the current position on a stack and returns via the pointer on the top of this stack. This is possibly because it was adapted from a recursive implementation.

The flush database in L4/MIPS consumes more memory than the page tables themselves. Furthermore, every page table update causes a costly search through a flush tree to find the corresponding flush node, and *vice versa*. The two structures are also difficult to make safe in a pre-emptable or SMP kernel.

6.2 Initial Optimization

The flush tree is a directed acyclic graph. If we perform a topological sort of this graph, there are many partial orderings, because each node can have a `child` and `sister` link. However, if we always resolve this choice by placing the `child` node first, a unique order is obtained. Figure 6.4 shows this topological sort order.

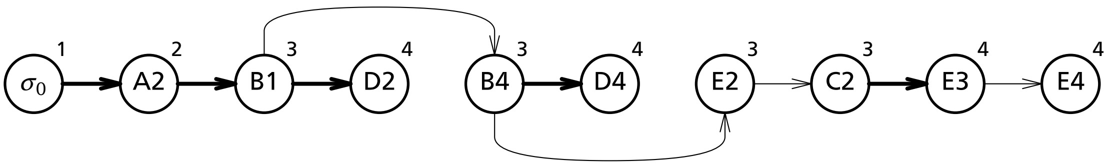

Figure 6.4 Topological sort of the above mapping graph — numbers indicate depth

Rather than including three links in each node, we instead string the nodes together with a single link in the above topological sort order (the flush order). All flushes proceed left-to-right along the list. To determine where to stop the flush, the most space-efficient solution is to add a small integer 'generation' number, recording the depth of the node in the equivalent flush tree. The flush algorithm then proceeds in flush order unlinking all the nodes of a younger generation than the root node, stopping when a node is reached of the same or older generation.

The advantage of this algorithm over the previous algorithm is that it is non-recursive and requires no stack. Compare this with the previous L4/MIPS implementation, which had three links per flush node, and the flush database of Schönberg and Uhlig [30] which has two links per flush node.

6.3 Flush and Super-Pages

Super-page support causes problems in the flush database, because large super-pages can be mapped in arbitrary size fragments. Once a super-page has been addressed as an f-page, there is no requirement that it need to be addressed as an f-page of the same size. Applications are free to map or flush a fragment of a large f-page, or incorporate any mixture of page sizes in a large f-page.

Figure 6.5 shows some address spaces composed with super-pages.

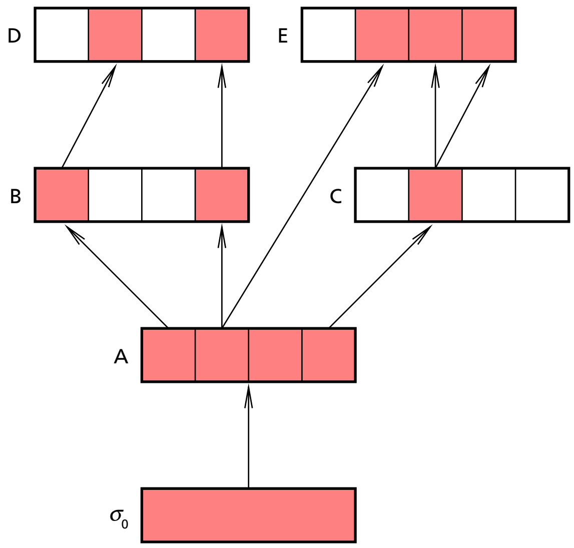

Figure 6.5 Address space composition with variable page sizes

The flush database must therefore be prepared to deal with arbitrary mixtures of page sizes.

Branching Flush Nodes

One solution would be to use an array of linked lists in each super-page node. This would require the flush node of a super-page to be larger than normal flush node. Also, an array does not cope well with a mixture of page sizes. For example, if a 1MB page was mapped in four 256kB fragments, the array would still have 256 elements representing 4kB fragments. A flush of a 256kB page would then be split up and flushed in 4kB pieces.

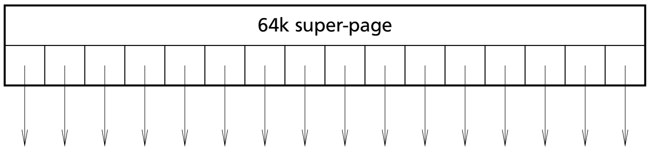

Figure 6.6 Flush node of a 64kB super-page mapped in 4kB fragments

A hierarchical structure could be used, such as a quadtree or even a full LPC trie. This would have the desirable property that the link from a super-page to a super-page of the same size would reduce to a single pointer. An application could then use constant page sizes throughout, and the kernel data structures would be identical to the equivalent using minimum page sizes.

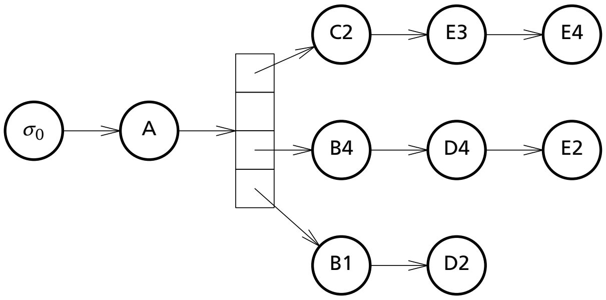

Figure 6.7 Flush lists with hierarchical branching

Expanded Flush Nodes

However, for simplicity, a branching strategy was not implemented in this version. Instead, all super-page keys are fully expanded in the flush database to the level of 4kB pages. Every flush node represents 4kB. Flushing a node which is a 4kB fragment of a larger page requires finding and splitting all the page table entries for the large page. Fortunately, only the update of each PTE needs to be atomic, and splitting a PTE requires the modification of only one double word.

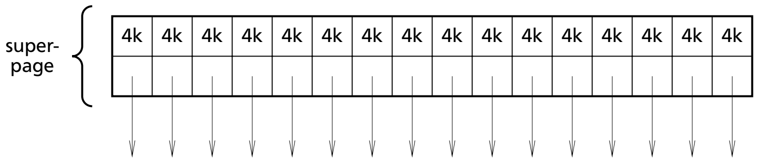

Figure 6.8 Fully expanded flush nodes which share a 64kB key

While this results in no saving in the space occupied by kernel page tables compared with the size before super-page support was added, it does not diminish the increase in TLB coverage possible with super-page support.

The decision was made to fully expand the nodes to solve race conditions in the memory allocator, and to make the addition of fine-grained locking simpler. Furthermore, expanding the keys decreases the fragmentation in the LPC trie caused by almost equal key lengths, and by the inability for tables to grow towards the low-order bits.

Removing the requirement that super-page keys be fully expanded is left for future work. It remains to be seen whether saving memory for super-pages justifies the additional complexity of implementing a branching flush database.

6.4 Merging Flush Lists into Page Table Entries

In the current L4/MIPS implementation all the flush trees are rooted in a frame table indexed by the physical frame number. To flush a page, the specified virtual address is looked up in the page table and a physical frame number obtained. This number indexes the frame table and the flush tree is searched to find the node to flush.

Apart from the inefficiency of funnelling all access to the flush database via the frame table, a linear frame table implicitly assumes a fixed page size, so is incompatible with the addition of super-page support.

Figure 6.9 is an overview of the important virtual memory data structures in L4/MIPS.

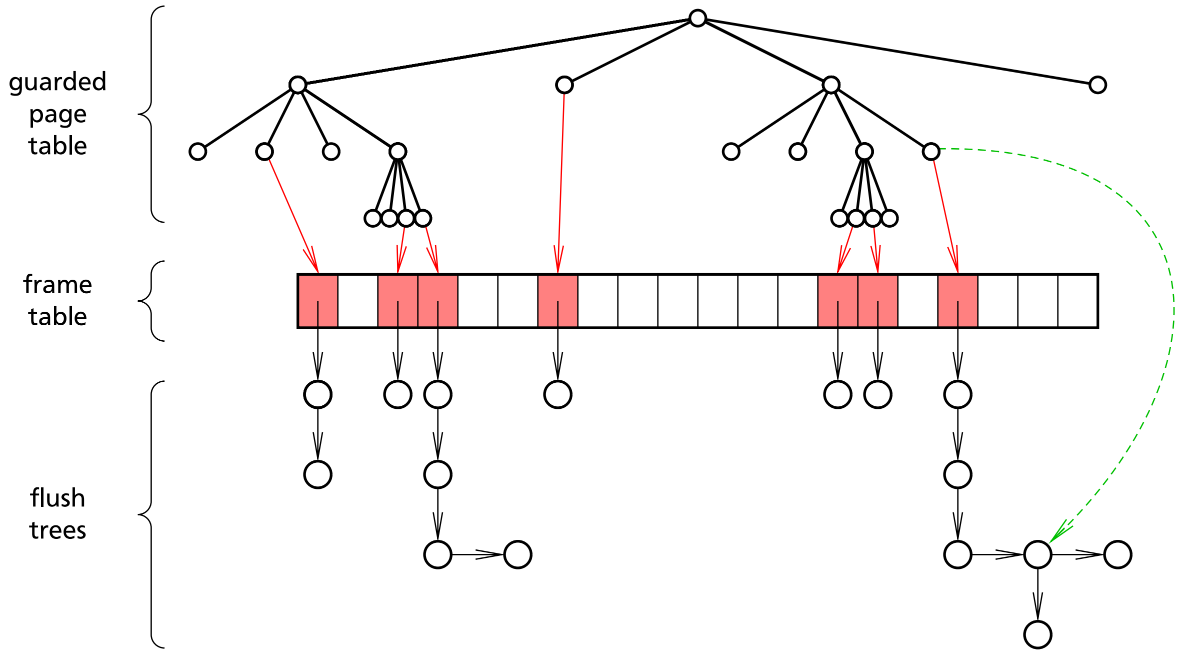

Figure 6.9 Current L4/MIPS virtual memory data structures

Each page table leaf corresponds to exactly one mapping node. It would be easier to store an additional pointer, to the mapping tree node, in each page table leaf. The dashed arrow (not part of L4/MIPS) shows how this additional pointer would bypass the frame table. The Alpha flush database of Schönberg and Uhlig [30] already does this.

However, adding a pointer from the page table entry to the flush node forces the size of the page table entry to exceed 16 bytes, especially since each PTE in the R4700 MMU represents a pair of pages. Furthermore, a flush node with only one small integer and one link as described above is only around 8 bytes in size.

Since the PTE must expand to the next power of two (32 bytes) to maintain the LPC trie's strict alignment requirements, the whole 8 byte flush node fits in the page table leaf, not just a 4 byte pointer to it. Figure 6.10 shows the full page table leaf format.

The large size of the flush node is a direct result of the R4700 page mask and paired TLB format. In an architecture without paired entries and less general super-page support, 16 bytes would suffice for external nodes even with the flush information.

This merging of flush information into the PTE makes sense when it is remembered that the virtual page number and page mask must be stored in the leaves of an LPC trie anyway to perform guard checking. This information is also present in the flush node, so with this merger the information can be stored once instead of twice.

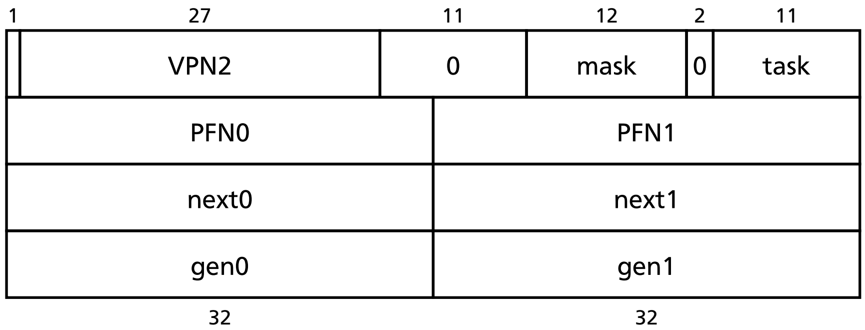

Figure 6.10 Format of combined PTE and flush node

Figure 6.11 shows how the flush lists fit into the LPC trie page table. Leaf nodes are contained within the page table levels, despite their larger size. The flush lists run through the leaf nodes, and are circular. This diagram is in fact simplified, because each leaf node in fact contains a pair of pages, and a pair of flush lists.

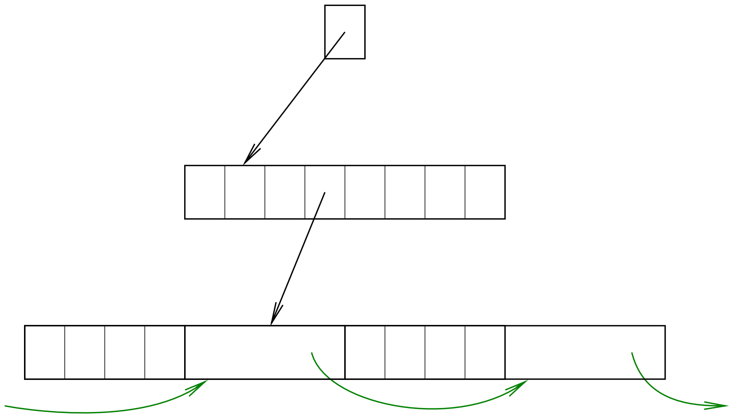

Figure 6.11 Flush lists running through the page table entries

Merging the flush links into the PTE reduces the amount of space overall, and makes the data structure simpler and easier to make pre-emptable. Furthermore, each node flushed occupies exactly one cache line, reducing the cache footprint of the flush algorithm.

6.5 Atomic Updates

Real-time Considerations

L4 is intended to be used to implement real-time systems. In real-time systems, there must be an upper bound on the interrupt response time. The time spent in the kernel with interrupts disabled or the interrupt priority level raised must be limited. Furthermore, real-time threads must be able to meet scheduling deadlines without being pre-empted by the kernel or non-real-time threads.

In L4, **hard priorities** are observed. High-priority threads are *always* scheduled before lower priority threads, so that a real-time thread with a deadline cannot be locked out of the CPU by lower priority threads. However, if a thread running in the kernel cannot be interrupted, a low priority thread can prevent a high priority thread from running by performing a long-running system call. In the worst case, a single flush system call in L4 can traverse every entry in every page table. It is therefore essential that long-running system calls are made pre-emptable.

Some real-time threads may need to perform L4 system calls in meeting their deadlines. For example, co-operating threads need to use inter-process communication (IPC) or have page faults serviced in normal operation. System calls which are expected to be used by real-time threads must also have predictable overall completion times.

Not every system call needs to be pre-emptable. System calls that are faster than the upper bound on interrupt response time need not be interruptible. For this reason, the L4 register-only IPC system call, which takes 102 cycles, is not pre-emptable.

Similarly, system calls that are not expected to be used by real-time threads need not have predictable completion times. However, if they run for a long time, they must be pre-emptable, so that non-real-time threads cannot prevent real-time threads from being scheduled.

SMP Considerations

L4 is intended to be used on symmetric multi-processor (SMP) systems. On SMP systems, kernel code cannot simply disable interrupts or raise the interrupt priority to protect critical sections, as the current interrupt priority level is local to the processor.

Schemes to lock other processors out of executing kernel code or to interrupt every other processor before and after each critical section are unacceptable, as the overall system performance will not scale to large numbers of processors. SMP kernels must therefore use locks on each data structure. The more fine-grained the locks are applied, the more scalable the SMP performance.

MIPS Locking Primitives

The primitives provided on MIPS for locking are the **load linked** and **store conditional** instructions. These instructions work just like conventional load and store, but the store conditional fails, zeroing the register to be stored, if the memory location was modified since the last load linked instruction.

Load linked and store conditional can be used to implement a spin-lock as follows:

```
spin:   ll      r0, (r1)
        bnez    r0, spin
        daddiu  r0, 1
        sc      r0, (r1)        /* sets r0 to zero on failure */
        beqz    r0, spin
```

Figure 6.12 Spin-locks implemented with load linked and store conditional instructions

Avoiding Locks

Spin-locks are unacceptable in a pre-emptable kernel, because a thread can be pre-empted while holding a lock, thus preventing a high-priority thread from manipulating the same data structure.

In the pre-emptable mapping database of Schönberg and Uhlig, if a lock is encountered during a flush, the thread donates its timeslice to the owner of the lock to help it release the lock. This leads to a priority inversion. For example, suppose a low priority thread is performing a long flush, and is pre-empted. If a high-priority thread performing a short flush operation encounters a lock held by the low priority thread, the low priority thread is given the remainder of the time-slice to help it release the lock. While this may only hold up the high priority thread for one timeslice, there is no upper bound on the number of times this can occur.

If an update can be interrupted in a half-finished state, then locks are required. However, if every update to any data structure is atomic, so that it will either complete successfully, or not succeed and leave the data structure untouched, then the data structure is always in a consistent state, and locks are not required.

This strategy can be used in the singly-linked flush database, because insertions always atomically add an element to a flush list following the specified node, and deletions consist of a sequence of unlink operations which can each be performed atomically.

Deletions are more difficult than in a doubly-linked list, but fortunately the most frequent L4 flush operation deletes the entries following the one specified in the flush. Therefore the address of the node before the specified node is not required unless the flush also removes the page in the flusher's own address space. This less common case is handled by tracing the circular linked list back to the beginning.

The following MIPS code atomically inserts the link `a1` directly after `a0` in a singly-linked list. `t0` and `t1` are temporary registers.

```
retry:  ll      t0, (a0)
        sw      t0, (a1)
        mov     t1, a1
        sc      t1, (a0)
        beqz    t1, retry
```

Figure 6.13 Atomic insert of an element into a singly-linked list

The following MIPS code unlinks the node pointed to by `a1` from its position after `a0`.

```
        ll      t0, (a0)
        bne     t0, a1, end
        lw      t1, (a1)
        sc      t1, (a0)
end:
```

Figure 6.14 Atomic unlink of an element into a singly-linked list

The final `sc` instruction fails either if a new node has been inserted between `a0` and `a1`, or if `a1` has been unlinked already. In the latter case, nothing needs to be done. In the former case, because the flush is proceeding linearly, the node will be encountered again.

The above functions are therefore customized for the map and flush operations. General lock-free atomic updates in linked lists [31] have been devised, requiring auxiliary nodes to allocated and atomically exchanged with the original node. The update is then performed only after the node's reference count is decremented to zero, indicating that it is referenced by no other thread.

7
Results

A prototype of the LPC trie page table implementation was compared with the GPT implementation from L4/MIPS. The GPT used a fixed node size of 16 entries. The LPC trie used a fixed allocation of 1MB of page table memory.

7.1 Page Table Depth

A number of address distributions were tested to determine the depth of both structures.

The key distributions were: a dense address blocks of 1MB, 4MB, and 16MB size, and a sparse distribution consisting blocks randomly sized between 4kB and 64kB, randomly distributed around a 4GB address space.

Figure 7.1 shows the average key depth of the two structures.

key distribution	GPT-16	LPC trie
1MB dense	2.0	1.0
4MB dense	3.0	1.0
16MB dense	3.0	1.0
1MB sparse	4.7	2.1
4MB sparse	5.4	2.9
16MB sparse	6.2	4.3

Figure 7.1 Average key depth for GPT-16 and LPC trie

Under increased load, the LPC trie implementation splits nodes more aggressively as page table memory is used up. This explains the deteriorating results slightly as the number of keys in the sparse benchmark increases. GPTs do better as the load increases, however, as sparsely occupied nodes are increasingly likely to be filled, and the nodes are never split. However, GPTs do not adapt when memory is scarce, and simply fail when no more memory is available.

7.2 Memory Usage

Level compression does have the potential for saving space, as the LPC trie omits internal nodes in dense regions of the address space.

A comparison of memory usage is difficult since the memory manager in this implementation deliberately allocates all of the available space. While the implementation uses tables as large as possible to maximize the amount of level compression, it records which entries are available to be split if necessary. The number of bits set in the bitmap for the minimum (64 byte) allocation size give extremely good results, as in principle nodes can be split to 64 bytes in size.

The benchmark chosen was the number of bits set in the bitmap for the 256 byte allocation size, since this size co-incides with the GPT-16 node size.

The GPT implementation manages its own memory by requesting memory one page at a time, and dividing it into 256 byte blocks for the page tables, and 24 byte blocks for the flush nodes. The estimates of GPT memory usage therefore overestimate the amount of memory used by about a page: half in free page tables, and half in free flush nodes.

Figure 7.2 shows the memory usage of the two structures.

key distribution	GPT-16	LPC trie
1MB dense	12	4
4MB dense	40	16
16MB dense	132	64
1MB sparse	40	6.25
4MB sparse	116	29.75
16MB sparse	392	143.25

Figure 7.2 Memory usage of GPT-16 and LPC trie in kilobytes

The feature which is crucial to effective level compression in the internal nodes is the minimum allocation size of 64 bytes. Further experiments are necessary to determine what choice of minimum allocation size is reasonable for different configurations.

8
Conclusions

8.1 Page Table Depth

Level compression can decrease the depth of page tables substantially for dense spaces. The LPC trie's behaviour for sparse address spaces is no worse than for the guarded page table. However, the split-only strategy used here is naïve about building optimal internal nodes. The small size of the internal nodes makes them a cheap target for more aggressive level compression. However, applying level compression more effectively to the internal nodes of a sparse page table requires the full complement of joining, doubling, and halving node primitives.

Strategies which integrate level compression with memory management have the potential to perform the best trade off between time and space.

Deferring the comparison of guards to the external nodes simplifies the messy handling of variable-length bit strings in the guarded page table, and reduces the number of cycles in the inner search loop. Mixing different size nodes in an LPC trie has proved easier than originally anticipated.

8.2 Super-Page Support

The R4700 page mask is a worthwhile feature which costs little to use. Super-pages can be expanded in the page table to whatever degree is necessary to reduce page table fragmentation, as the benefit of super-pages is in the increased coverage of the TLB.

However, in contrast to the experience of Talluri [30] the operating system support for the paired R4700 TLB format is more involved than the support for super-pages. Furthermore, paired super-pages are less likely to be used by applications.

A major benefit of super-page support in L4 is the reduction in the TLB footprint of initial server applications which have σ_0 as their pager. Such applications would frequently map out these super-pages in much smaller fragments to user applications, thus justifying the complete key expansion technique used in this implementation.

The drawback of this implementation is that super-pages do not save page table memory as they can in some hierarchical page table structures. Removing the restriction that keys must be expanded completely is left for future work.

Because it maps the initial address space to initial servers in 4kB pages, the present L4/MIPS API does not allow address spaces other than σ_0 to make use of super-pages. This will require change in a future version of L4/MIPS.

8.3 TLB Refill Path

While complete benchmarks are presently unavailable, it would seem that the best TLB refill routine is a direct-mapped TLB cache without super-page support backed by a LPC trie with full super-page support. Any modification of the TLB cache refill routine to add features to reduce TLB cache conflicts will probably negate the principle benefit of the TLB cache, which is the low cost of refills which hit in the TLB cache.

Further measurements of the utilization of super-pages, the frequency of TLB misses, and the frequency of TLB cache misses must be made. If the cost of TLB cache misses is too great for workloads where frequent address space switching occurs, the TLB cache may have to abandoned in favour of an LPC trie only refill routine.

8.4 L4 Flush Lists

The L4/MIPS flush database has a straightforward representation as a linked list. This representation is obtained directly from graph theory. The linked list representation is more efficient overall and much easier to make safe for pre-emptability.

Merging the flush lists into the page table entries reduces the overall complexity and demands on the memory manager. Frequently map operations require no extra memory to be allocated at all.

Pre-emptable data structures require much more thought and planning than the same data structures in a non-pre-emptable system. Use of lock-free primitives requires much more work than simply adding a spin-lock to each data structure, as each update has to be examined for consistency, and each update needs code to handle failure.

Retro-fitting an existing structure with pre-emptability is much more difficult than designing a structure with pre-emptability in the first place.

References

[1] Arne Andersson, Stefan Nilsson, *Improved Behaviour of Tries by Adaptive Branching*, Information Processing Letters, 46:295–300, 1993. http://www.nada.kth.se/~snilsson/public/-papers/impr/text.pdf

[2] Jochen Liedtke, *Page Table Structures for Fine-Grain Virtual Memory*, in IEEE Technical Committee on Computer Architecture Newsletter, 1994.

[3] Kevin Elphinstone, Gernot Heiser, *L4 Reference Manual*, UNSW-CSE-TR-9709, School of Computer Science and Engineering, University of New South Wales, December 1997. http://www.cse.unsw.edu.au/~disy/L4/MIPS/l4uman.ps.gz

[4] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, Jochen Liedtke, *Implementation and Performance of the Mungi Single-Address Space Operating System*, Software Practice and Experience, 28(9):901–928, July 1998. ftp://ftp.cse.unsw.edu.au/pub/users/disy/-papers/Heiser_EVRL_98.ps.gz

[5] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, Edward D. Lazowska, *Sharing and Protection in a Single-Address-Space Operating System*, ACM Transactions on Computer Systems, 12(4):271–307, November 1994. ftp://ftp.cs.washington.edu/tr/1993/04/UW-CSE-93-04-02.PS.Z

[6] J. Fotheringham, *Dynamic Storage Allocation in the Atlas Computer, Including an Automatic Use of a Backing Store*, Communications of the ACM, 4(10), October 1961.

[7] John Cocke, *Virtual to Real Address Translation Using Hashing*, IBM Technical Disclosure Bulletin, 24(6), November 1981.

[8] R. P. Case, A. Padegs, *Architecture of the IBM System/370*, Communications of the ACM, 21(1):73–96, 1978.

[9] H. M. Levy, P. H. Lipman, *Virtual Memory Management in the VAX/VMS Operating System*, IEEE Computer, 15(3):35–41, March 1982.

[10] S. A. Ritchie, *TLB for Free: In-Cache Translation for a Multi-processor Workstation*, Berkeley University Report UCB/CSD 85/233, 1985.

[11] R. G. Hintz, D. P. Tate, *Control Data STAR-100 Processor Design*, COMPCON '72 Digest, 1972.

[12] J. L. Hennessy, N. Jouppi, F. Baskett, J. Gill, *MIPS: A VLSI Processor Architecture*, Proceedings of the CMU Conference on VLSI Systems and Computations, pp. 337–346, 1981.

[13] D. A. Huffman, *A method for the construction of minimum redundancy codes*, Proceedings of IRE, 40:1098–1101, 1952.

[14] Intel Corporation, *Intel Architecture Software Developer's Manual*, *Volume 3: System Programming*, Intel Order Number 243192, August 1999. ftp://download.intel.com/design/-pentiumii/manuals/24319202.pdf

[15] Robert R. Collins, *Paging Extensions for the Pentium Pro Processor*, July 1996. http://www.ddj.com/articles/1996/9607/9607m/9607m.htm

[16] Donald R. Morrison, *PATRICIA — Practical Algorithm To Retrieve Information Coded In Alphanumeric*, Journal of the ACM, 15(4):514–534, October 1968.

[17] Donald E. Knuth, *The Art of Computer Programming*, volume 3, *Sorting and Searching*. Addison-Wesley, 1973.

[18] Keith Sklower, *A Tree-Based Packet Routing Table for Berkeley Unix*, USENIX Association Conference Proceedings, pp. 93–99, January 1991. http://www.cs.berkeley.edu/~sklower/-routing.ps

[19] Arne Andersson, Stefan Nilsson, *Faster Searching in Tries and Quadtrees — An Analysis of Level Compression*, Proceedings of Second Annual European Symposium on Algorithms, pp. 82–93, 1994. http://www.nada.kth.se/~snilsson/public/pap-ers/fast/text.pdf

[20] Stefan Nilsson, Matti Tikkanen, *Implementing a dynamic compressed trie*, Proceedings of the Second Workshop on Algorithm Engineering, Saarbrücken, Germany, pp. 20–2, August 1998. http://www.nada.kth.se/~snilsson/public/papers/dyntrie/-text.pdf

[21] Donald E. Knuth, *The Art of Computer Programming*, volume 1. Addison-Wesley, 1973.

[22] Gerry Kane, Joe Heinrich, *MIPS RISC Architecture*, Prentice Hall, 1992.

[23] Integrated Device Technology, *IDT79R4600 and IDT79R4700 RISC Processor Hardware User's Manual*, April 1995.

[24] Kevin Elphinstone *et al.*, L4/MIPS micro-kernel distribution, version 79, 1999. ftp://ftp.cse.unsw.edu.au/pub/users/disy/mipsL4/mipsL4-79.tar.gz

[25] Jochen Liedtke, Kevin Elphinstone, *Guarded Page Tables on the MIPS R4600*, UNSW-CSE-TR-9503, School of Computer Science and Engineering, University of New South Wales, November 1995. ftp://ftp.cse.unsw.edu.au/pub/doc/papers/-UNSW/9503.ps.Z

[26] Kavita Bala, M. Frans Kaashoek, William E. Weihl, *Software Prefetching and Caching for Translation Lookaside Buffers*, Proceedings of the First Symposium on Operating System Design and Implementation (OSDI), pp. 243–253, Novermber 1994.

[27] Jerry Huck, Jim Hays, *Architectural Support for Translation Table Management in Large Address Space Machines*, Proceedings of the 20th Annual International Symposium on Computer Architecture, pp. 39–50, May 1993.

[28] Kevin Elphinstone, *Virtual Memory in a 64-bit Micro-kernel*, PhD Thesis, UNSW, 1999. ftp://ftp.cse.unsw.edu.au/pub/users/disy/papers/Elphinstone:phd.ps.gz

[29] Sebastian Schönberg, *The L4 micro-kernel on Alpha — Design and Implementation*, Cambridge University Technical Report #407, 1996.

[30] Sebastian Schönberg, Volkmar Uhlig, *μ-kernel Memory Management*, Dresden University of Technology Technical Report, May 1996. http://os.inf.tu-dres-den.de/~vu2/techreport.ps

[31] John D. Valois, *Lock-Free Linked Lists Using Compare-and-Swap*, Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, p. 214, 1995. http://www.acm.org/pubs/articles/proceedings/podc/224964/p214-valois/-p214-valois.pdf

[32] Madhusudhan Talluri, Mark D. Hill and Yousef A. Khalidi, *A New Page Table for 64-bit Address Spaces*, Proceedings of 15th ACM Symposium on Operating System Principles, December 1995. ftp://ftp.cs.wisc.edu/markhill/Papers/sosp95_page-tables.ps

[33] Madhusudhan Talluri, *Use of Super-pages and Sub-blocking in the Address Translation Hierarchy*, Computer Sciences TR #1277, University of Wisconsin. http://www.cs.wisc.edu/~talluri/thesis/thesis.tr.ps

[34] Motorola Incorporated, *MC68851 Paged Memory Management Unit User's Manual*, 1986.